

From Rewriting Logic Executable Semantics to Matching Logic Program Verification

Grigore Roşu

University of Illinois at
Urbana-Champaign
grosu@illinois.edu

Chucky Ellison

University of Illinois at
Urbana-Champaign
celliso2@illinois.edu

Wolfram Schulte

Microsoft Research
Redmond
schulte@microsoft.com

Abstract

Rewriting logic semantics (RLS) is a definitional framework in which a programming language is defined as a rewrite theory: the algebraic signature defines the program configurations, the equations define structural identities on configurations, and the rewrite rules define the irreversible computational steps. RLS language definitions are efficiently executable using conventional rewrite engines, yielding interpreters for the defined languages for free.

Matching logic is a program verification logic inspired by RLS. Matching logic specifications are particular first-order formulae with constrained algebraic structure, called *patterns*. Configurations satisfy patterns iff they match their algebraic structure and satisfy their constraints. Patterns can naturally specify data separation and require no special support from the underlying logic.

Using HIMP, a C-like language with dynamic memory allocation/deallocation and pointer arithmetic, this paper shows how one can derive an executable matching logic verifier from HIMP's RLS. It is shown that the derived verifier is sound, that is every verified formula holds in the original, complementary RLS of HIMP, and complete, that is every verified formula is provable using HIMP's sound matching logic proof system. In passing, this paper also shows that, for the restriction of HIMP without a heap called IMP for which one can give a conventional Hoare logic proof system, a restricted use of the matching logic proof system is equivalent to the Hoare logic proof system, in that any proof derived using any of the proof systems can be turned into a proof using the other. The encoding from Hoare logic into matching logic is generic and should work for any Hoare logic proof system.

A matching logic verifier, called `MATCHC`, has been built on top of the Maude rewrite system. A nontrivial `MATCHC` case study is discussed, namely the verification of the partial correctness of the Schorr-Waite algorithm (with graphs). The verifier automatically generated and proved all 227 paths in 16 seconds.

1. Introduction

Program language semantics and program verification are well developed research areas with a long history. In fact, one might think that all problems would have been solved by now: we would hope that any formal semantics for an imperative language should give

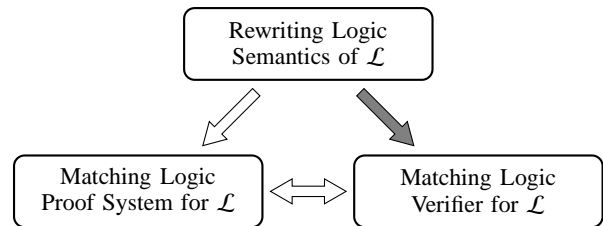


Figure 1. Overview of our verification approach for language \mathcal{L}

rise to a proof system and that a verifier for such a system would simply extend the proof system with a proof strategy; or looked at from the other side, we would assume that any verification system for a particular programming language would be grounded in that language's formal semantics. However reality tells us that building in semantics grounded verifiers for realistic languages is still a dream. Popular languages, like C, C# or Java had initially no formal semantics, just reference implementations or informal reference manuals. Their corresponding verifiers like Caduceus [Filliâtre and Marché 2004] or VCC [Cohen et al. 2009] for C, Spec# [Barnett et al. 2004] for C#, or ESC/Java [Flanagan et al. 2002] for Java, are best efforts to capture the behavior of a particular implementation or of an informal reference manual. But how can the verification community claim to do serious verification on real programs, if we don't relate verification systems to semantics and semantics to implementations? This paper tries to address the first of these issues: it links program language semantics and program verification formally, as shown in Figure 1. We proceed in three steps:

First, we suggest using *Rewriting Logic Semantics* (RLS) as a flexible and expressive logical framework to give meaning to programs. RLS defines the meaning of programs as a rewrite theory: the algebraic signature defines the program *configurations*, like code fragments to be executed, environments and stores; the equations define *structural identities* on configurations, e.g. that execution proceeds from left to right or that the order of variable-value pairs in the environment does not matter; and the *rewrite rules* define the irreversible computational steps, such as an assignment updates the environment. Such semantic definitions are directly executable as interpreters in a rewriting system such as Maude [Clavel et al. 2007]. RLS scales well; even though we exemplify RLS by defining a simple language in this paper, it has in fact been used to define real languages, like Java 1.4 [Farzan et al. 2004].

Second, we introduce a provably sound *Matching Logic* proof system for a given RLS. Matching logic is similar to Hoare logic in many aspects. Like Hoare logic, matching logic specifies program states as logical formulae and gives an axiomatic semantics

to a programming language in terms of pre- and post-conditions. Like Hoare logic, matching logic can generically be extended to a formal, syntax-oriented compositional proof system. However, unlike Hoare logic, matching logic configurations will not be flattened to arbitrary first order logic (FOL) formulas; instead they are kept as symbolic configurations, i.e. restricted FOL₌ (FOL with equality) formulae. Matching logic specifications, e.g. pre- and postconditions, are *patterns* over configurations, possibly containing both free and bound variables. A configuration *matches* a pattern iff it is obtained as an instance of the pattern (i.e., mapping pattern's variables to concrete values). Matching logic, which was only introduced this year in a technical report [Roşu and Schulte 2009], has unique benefits compared to other program logics, for instance:

- Matching logic, by means of its patterns, separates the syntactic ingredients, such as the program variables and expressions, from the semantic ones, such as the logical variables; it thus eliminates the problems and limitations due to mixing program syntax with logical specifications.
- Matching logic achieves *heap separation* without having to extend the logic with special connectives; e.g. the very fact that one can match two trees in a heap means, by definition, that the two trees are separate; in other words, unlike in separation logic, in matching logic separation is nothing special and it is achieved at the term level instead of at the formula level; in particular, no disjointness predicate is needed.
- Matching logic, by means of its algebraic foundation, often allows to substitute *algebraic axiomatizations* of uninterpreted operations for recursive predicates which are a problem for FOL. For example, the operation reverse on lists can be axiomatized with two equations, $rev(nil) = nil$ and $rev(a:\alpha) = rev(\alpha):a$, where list concatenation (here written with the infix operator “:”) is defined equationally as an associative and commutative operator. Tools, such as Maude, which can execute RLS and can be used to build program verifiers based on matching logic, are quite effective in using those.

Third, and perhaps more importantly from a practical perspective, executable and thus testable RLS language definitions can easily be turned into efficiently executable automatic *Matching Logic Verifiers* based on symbolic *forward simulation*, which are sound and complete for the matching logic axiomatic semantics. The following language-independent steps need to be taken to turn a RLS into such a matching logic verifier: (1) extend RLS configurations with new configuration items that keep track of bound variables and path conditions; this extension is modular, in that none of the existing equations and rules in the original RLS need to change; (2) add an explicit case-split rule to deal with conditional control structures and keep track of path conditions (this rule corresponds to a similar sound matching logic rule); (3) distinguish successful proofs from failed proof attempts. In addition to the above background infrastructure that can in principle be generated automatically for any existing RLS, the semantics of control and memory allocation constructs in the original RLS has to be adapted to work on symbolic configurations. Interestingly, as the steps above may suggest, one can derive a matching logic verifier from an RLS of a language more easily than one can derive a matching logic proof system, which is the reason why we emphasized the corresponding arrow in Figure 1. The matching logic proof system is, nevertheless, worthwhile for many reasons: it rigorously justifies the correctness of our verifier; it serves as a foundational backbone for other verifiers, for example ones based on backwards analysis; etc.

Matching logic verifiers maintain a clean separation between the property to check, the program and the language semantics. This makes *debugging* failed verification attempts easy: when the verifier gets stuck, it simply shows the current configuration — it has

all the information that is needed to understand what has happened: the code where the prover got stuck, the path which had been taken, the assumptions on environment and store, etc. Compare this simplicity to the intricate details that a modern weakest-precondition-based prover like Boogie [Barnett et al. 2006] has to maintain.

We demonstrate these three steps by deriving a matching logic verifier for HIMP, a C-like language with dynamic memory allocation/deallocation and pointer arithmetic. We provide a theorem for the soundness of the matching logic formal system w.r.t. the original, complementary and testable RLS semantics; and a soundness and completeness theorem for the correctness of the verifier w.r.t. the matching logic proof system. To show the practicality of the approach, we implemented an extended version of the derived matching logic verifier, called MATCHC, using a modified version of Maude that invokes off-the-shelf SMT solvers to discharge proof obligations that cannot be solved by rewriting. We demonstrate the effectiveness of MATCHC by applying it on a nontrivial case study: the partial correctness of the Schorr-Waite algorithm (with graphs). Our formalization uses novel axiomatizations of clean and marked partial graphs, as well as a specific stack-in-graph structure, which records that during the execution of Schorr Waite the heap consists at any time of such a stack and either a clean or a marked partial subgraph. With this formalization the prover automatically generated and verified all paths (over 200) in a few seconds. To the best of our knowledge, this is the first time the partial correctness of this algorithm has been automatically verified. Previous efforts have either proved only its memory safety [Hubert and Marche 2005] or a version restricted to trees [Loginov et al. 2006] automatically.

The novel contributions of this paper are thus as follows:

- We provide a rewriting logic semantics for HIMP, a C-like language with dynamic memory allocation/deallocation and pointer arithmetic. We also give the fragment of HIMP without a heap, called IMP, a Hoare logic axiomatic semantics as a proof system and show that it is sound w.r.t. the RLS of IMP.
- We introduce matching logic and show how a matching logic proof system can be derived from an existing RLS of a language, here HIMP. We prove the soundness of the matching logic proof system w.r.t. the original RLS, first for IMP and then for HIMP.
- We show that for IMP, a restricted use of the matching logic proof system is equivalent, via a back-and-forth mechanical translation, to a conventional Hoare logic proof system for IMP. The translation from Hoare logic to matching logic is generic and should work for any language, suggesting that any Hoare logic proof system admits an equivalent matching logic proof system. The other translation, from matching logic to Hoare logic, appears to be language specific, because it relies on finding appropriate encodings of program configuration patterns into Hoare specifications; it is not clear that they always exist.
- We derive a matching logic verifier for HIMP from its RLS and show that it is sound and complete for the matching logic proof system of HIMP. That means, in particular, that the derived program verifier for HIMP is sound w.r.t. the original executable (and thus debuggable) complementary RLS of the language.
- We briefly introduce MATCHC, a matching logic verifier for KERNELC (an extension of HIMP with `malloc` and `free`), which has been built on top of the Maude rewriting system. It efficiently and automatically proves the partial correctness of the Schorr-Waite algorithm with graphs (invariant is provided).
- We provide a new and simple approach to prove Schorr-Waite, based on axiomatizations of clean and marked graphs, as well as of the specific stack-in-graph structure, and provide an invariant stating that the heap consists at any time of such a stack and either a clean or a marked partial subgraph.

Section 2 introduces some background notions and notation. Section 3 gives the RLS semantics of IMP, the variant of HIMP without a heap, gives a Hoare logic axiomatic semantics of IMP as a proof system, and finally shows that the latter is sound w.r.t. the former. Section 4 introduces some generic matching logic notions, then gives a sound matching logic axiomatic semantics to IMP as a proof system, and then finally shows that a restricted use of this matching logic proof system is equivalent to the Hoare logic proof system of IMP. Section 5 shows how to derive a sound and complete matching logic prover for IMP. Section 6 shows how easily the concepts and techniques defined for IMP extend to HIMP; we show how several heap patterns are also axiomatized, such as lists, trees and graphs. Section 7 briefly discusses our implementation `MATCHC` and shows how we used it to prove the partial correctness of the Schorr-Waite algorithm.

2. Preliminaries

We assume the reader is familiar with basic concepts of multi-sorted algebraic specification, rewriting logic, and first-order logic with equality. The role of this section is to establish our notation for concepts and notions used later in the paper.

An *algebraic signature* (\mathbb{S}, Σ) consists of a finite set of *sorts* \mathbb{S} and of a finite set of *operation symbols* Σ over sorts in \mathbb{S} . For example, in the context of a programming language, \mathbb{S} may include sorts like E for expressions and S for statements, and Σ may include operation symbols like `if()_else_` : $E \times S \times S \rightarrow S$, where the underscores are placeholders for the corresponding arguments; this style of writing operation symbols using underscores is called *mixfix* and is supported by several languages in the OBJ family [Goguen et al. 2000]. Many-sorted algebraic signatures like above are equivalent to context-free grammars (CFG): sorts correspond to non-terminals and mixfix operation symbols to productions; e.g., “`if()_else_` : $E \times S \times S \rightarrow S$ ” corresponds to “ $S ::= \text{if}(E) S \text{ else } S$ ”. From here on we prefer to use the CFG notation for algebraic signatures, because it is more common in the context of programming languages. We may write Σ instead of (\mathbb{S}, Σ) when \mathbb{S} is understood or when it is irrelevant. Signatures can be used to build terms, which can be organized into an initial algebra. We let \mathcal{T}_Σ denote the *initial Σ -algebra of ground terms* (i.e., terms without variables) and let $\mathcal{T}_\Sigma(X)$ denote the *free Σ -algebra of terms with variables in X* , where X is an \mathbb{S} -indexed set of variables.

An *algebraic specification* (Σ, \mathcal{E}) consists of an algebraic signature Σ and a set of Σ -equations \mathcal{E} , where a Σ -equation is a triple $\forall X.(t = t')$, where X is a set of variables and t, t' are terms having the same sort in $\mathcal{T}_\Sigma(X)$. Algebraic specifications (Σ, \mathcal{E}) admit initial and free models, obtained by factoring the initial and free term models by all equational consequences of \mathcal{E} . Many domains of interest to languages, such as integer numbers, finite lists (sequences), sets and multisets (bags), mappings of finite domain, etc., can be axiomatized as initial models of finite algebraic specifications.

A *rewriting logic specification* [Meseguer 1992] $(\Sigma, \mathcal{E}, \mathcal{R})$ adds to an algebraic specification (Σ, \mathcal{E}) a set of *rewrite rules* of the form $\forall X.(l \rightarrow r)$, where X is a set of variables and l, r are terms of the same sort in $\mathcal{T}_\Sigma(X)$. Rewriting logic specifications can be used to define dynamic and concurrent systems: equations are regarded as structural identities *modulo* which the irreversible rewrite rules are applied. It was shown [Şerbănuță et al. 2009] that rewriting logic embeds various programming language definitional styles, such as structural operational semantics [Plotkin 2004], reduction semantics with evaluation contexts [Wright and Felleisen 1994], the CHAM [Berry and Boudol 1992], continuations [Reynolds 1993], etc. By “embed”, as opposed to “encode”, is meant that the corresponding rewriting logic specifications capture the original semantics step-for-step, i.e., they do not change the computational granularity of the embedded semantics. \mathbb{K} [Roşu 2007, Meseguer and

Roşu 2007] is a rewriting logic language definitional technique employed in this paper to give executable semantics to our language(s). \mathbb{K} overcomes some of the limitations of the above-mentioned styles and fully exploits and extends the strengths of rewriting logic.

We next briefly recall *first-order logic with equality* ($FOL_=_$). A *first-order signature* $(\mathbb{S}, \Sigma, \Pi)$ extends an algebraic signature (\mathbb{S}, Σ) with a finite set of predicate symbols Π ; we also use the mixfix notation for predicates, e.g., $<_< : E \times E$, etc. $FOL_=_$ formulae have the syntax $\varphi ::= t = t' \mid \pi(\vec{t}) \mid \exists X.(\varphi) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2$, plus the usual derived constructs $\varphi_1 \vee \varphi_2$, $\varphi_1 \Rightarrow \varphi_2$, $\forall X.(\varphi)$, etc., where t, t' range over Σ -terms of the same sort, $\pi \in \Pi$ over atomic predicates and \vec{t} over appropriate term tuples, and X over finite sets of variables. Σ -terms can have variables; all variables are chosen from a fixed sort-wise infinite \mathbb{S} -indexed set of variables, *Var*. We adopt the conventional notions of *free* and *bound* variables in formulae, and the notation $\varphi[e/x]$ for the capture-free substitution of all free occurrences of variable x by term e of same sort in formula φ . A *$FOL_=_$ specification* $(\mathbb{S}, \Sigma, \Pi, \mathcal{F})$ is a $FOL_=_$ signature $(\mathbb{S}, \Sigma, \Pi)$ plus a set of *closed* (i.e., no free variables) formulae \mathcal{F} . A *$FOL_=_$ model* M is a Σ algebra together with relations for the predicate symbols in Π . Given any closed formula φ and any model M , we write $M \models \varphi$ iff M satisfies φ . If φ has free variables and $\rho : \text{Var} \rightarrow M$ is a partial mapping defined (at least) on all free variables in φ , also called an *M -valuation* of φ 's free variables, we let $\rho \models \varphi$ denote the fact that ρ satisfies φ . Note that $\rho \models \varphi[e/x]$ iff $\rho[\rho(e)/x] \models \varphi$.

3. A Simple Imperative Language Without Heap

This section introduces (our version of) IMP, a simple imperative language with assignments, conditionals and while loops. Like C, IMP has no boolean expressions (0 means “false” and $\neq 0$ means “true”) but unlike C, IMP uses `:=` instead of `=` for assignments (to avoid confusion with `=` of $FOL_=_$). We give IMP an executable semantics in the \mathbb{K} rewrite framework, and an axiomatic semantics as a Hoare logic proof system; the latter is also shown sound w.r.t. the former. Both of these semantics will be needed later in the paper. Section 6 extends IMP with memory allocation and pointers (including pointer arithmetic); the new language will be called HIMP (IMP with a heap).

3.1 \mathbb{K} Executable Definition of IMP

\mathbb{K} is an executable rewriting logic language definition framework in which a programming language \mathcal{L} is defined as a rewrite logic theory $(\Sigma_{\mathcal{L}}, \mathcal{E}_{\mathcal{L}}, \mathcal{R}_{\mathcal{L}})$. $\Sigma_{\mathcal{L}}$ includes the syntax of \mathcal{L} and additional syntax needed to define configurations and the actual language semantics, $\mathcal{E}_{\mathcal{L}}$ includes structural equations that have no computational meaning but can “rearrange” terms so that semantic rules match and apply, and $\mathcal{R}_{\mathcal{L}}$ includes irreversible rules that correspond to intended computational steps in the defined language.

Figure 2 shows the complete \mathbb{K} definition of IMP. The signature is given via CFG notation and consists of the IMP syntax plus the syntax of configurations. \mathbb{K} makes use of bags (or multisets), sets, sequences and finite maps to define configurations. All of these structures are routinely defined equationally and our implementation of \mathbb{K} automatically generates Maude algebraic specifications for them whenever used. Each of these structures admit a unit and a binary construct; these are given as subscript and superscript, respectively. Maps have additional polymorphic constructs for pairs, namely $_ \mapsto _$, and for update, namely $_[-/-]$. For example, the *Env* sort is defined in Figure 2 as $\text{Map}^{\text{--}}[PVar, Int]$, that is, environments are map structures of the form “ $x_1 \mapsto i_1, x_2 \mapsto i_2, \dots, x_n \mapsto i_n$ ”, with “ $_$ ” representing the empty environment. Technically, it means that several operations and equations are added: “ $_ _ : Env \times Env \rightarrow Env$ ” as a binary associative and commutative construct with “ $_$ ” as unit, “ $_ \mapsto _ : PVar \times Int \rightarrow Env$ ” as atomic environment construct, and “ $_[-/-] : Env \times PVar \rightarrow Int$ ” and “ $_[-/-] : Env \times Int \times PVar \rightarrow Env$ ”

$Nat ::= \text{naturals}, Nat^+ ::= \text{pos. naturals}, Int ::= \text{integers}$	(abstract syntax)
$PVar ::= \text{identifiers, to be used as program variable names}$	
$E ::= Int \mid PVar \mid E_1 \text{ op } E_2$	
$S ::= PVar := E \mid S_1 ; S_2 \mid \text{if } (E) S_1 \text{ else } S_2 \mid \text{while } (E) S$	
$Cfg ::= \langle Bag[CfgItem] \rangle$	(configuration)
$CfgItem ::= \langle K \rangle_k \mid \langle Env \rangle_{env}$	
$K ::= E \mid S \mid Seq \sim [K] \mid \square$	
$Env ::= Map \cdot [PVar, Int]$	
$(x := e) = (e \leadsto x := \square)$	(structural “strictness” equations)
$e_1 \text{ op } e_2 = (e_1 \leadsto \square \text{ op } e_2) \quad i_1 \text{ op } e_2 = (e_2 \leadsto i_1 \text{ op } \square)$	
$\text{if } (e) s_1 \text{ else } s_2 = (e \leadsto \text{if } (\square) s_1 \text{ else } s_2)$	
$s_1 ; s_2 = s_1 \leadsto s_2$	(semantic equations and rules)
$i_1 \text{ op } i_2 \rightarrow i_1 \text{ op}_{Int} i_2$	
$\langle x \leadsto k \rangle_k \langle x \mapsto i, \rho \rangle_{env} \rightarrow \langle i \leadsto k \rangle_k \langle x \mapsto i, \rho \rangle_{env}$	
$\langle x := i \leadsto k \rangle_k \langle \rho \rangle_{env} \rightarrow \langle k \rangle_k \langle \rho[i/x] \rangle_{env}$	
$\text{if } (i) s_1 \text{ else } s_2 \rightarrow s_1,$	where $i \neq 0$
$\text{if } (0) s_1 \text{ else } s_2 \rightarrow s_2$	
$\langle \text{while } (e) s \leadsto k \rangle_k = \langle (\text{if } (e) s ; \text{while } (e) s \text{ else } \cdot) \leadsto k \rangle_k$	
$(x \in PVar; e, e_1, e_2 \in E; s, s_1, s_2 \in S; k \in K; i, i_1, i_2 \in Int; \rho \in Map \cdot [PVar, Int])$	

Figure 2. IMP in \mathbb{K} : Complete Executable Semantics

as environment lookup and update operations; in addition to the obvious equations defining all the above, constraints ensuring that $x_i \neq x_j$ whenever $i \neq j$ are also added.

Configurations in \mathbb{K} are defined as potentially nested structures of such bags, sets, sequences and maps; these structures are called *cells* and are labeled to avoid confusion. The configurations of our simple IMP language have the form $\langle \dots \rangle_k \langle \dots \rangle_{env}$, so are bag cells containing two subcells — a computation (which is a sequence cell) and an environment (which is a map cell). We add a new heap cell when we extend IMP to HIMP in Section 6. Configurations can be arbitrarily complex and their cells can be created and destroyed dynamically. For example, the configurations of our Java 1.4 definition [Farzan et al. 2004] contain nine fixed top-level cells plus one other cell per dynamically created thread, each such thread cell containing eight other subcells holding the thread-specific data.

A distinctive aspect of \mathbb{K} is the K sort and its constructs. The K sort stands for *computational structures*, or simply *computations*. Computations sequentialize computational tasks separated by an associative \sim (read “then”, or “followed by”). The placeholder “ \square ” signifies where the result of the previous computation should be plugged.¹ Recall that in rewriting logic and implicitly in \mathbb{K} , equations are reversible and rewrite rules are applied modulo equations. For example, the equation “ $(x := e) = (e \leadsto x := \square)$ ” states that the assignment “ $x := e$ ” is to be considered identical to the computation “ $e \leadsto x := \square$ ”, which schedules e for processing and says how its result will be plugged back into the right assignment context. This equation will need to be applied twice, once to unplug e and schedule it for processing, and another to plug the result of e , once computed, back into the assignment statement. Structural equations define the evaluation strictness of the language constructs and correspond to the definition of the grammar of evaluation contexts in reduction semantics [Wright and Felleisen 1994]; however, the process of parsing a term into a context and a redex in reduction semantics is replaced by equational deduction in \mathbb{K} .

The semantic equations and rules are the core of a \mathbb{K} semantics, with typically one per language construct (two or more only for choice statements). For IMP, there are two semantic equations:

¹ The meaning of \square in \mathbb{K} in its full generality is slightly more complex, but we do not need its full generality here.

one for desugaring the sequential composition and one for the while loop unrolling. We made these equations because we do not want them to count as computational steps, but one is free to make them rules instead. The domains of concrete values typically come with suitable signatures that one can use in the semantics; e.g., the IMP “ $+$ ” expression construct is reduced to the domain “ $+_{Int} : Int \times Int \rightarrow Int$ ” when its arguments become values (integers in this case). Note that the while loop is unrolled only when it is the top task in the computation, to avoid non-termination of unrolling. The semantic rewrite rules are self-explanatory; we only discuss the assignment rule “ $\langle x := i \leadsto k \rangle_k \langle \rho \rangle_{env} \rightarrow \langle k \rangle_k \langle \rho[i/x] \rangle_{env}$ ”. This says that if the assignment “ $x := i$ ” is the top computational task and the current environment is ρ , then remove the assignment from the computation and update the environment with i for x .

We can now formally define IMP as a rewriting logic specification/theory, together with some terminology.

DEFINITION 1. We let IMP denote the rewriting logic specification $(\Sigma_{IMP}, \mathcal{E}_{IMP}, \mathcal{R}_{IMP})$ in Figure 2 and, like in rewriting logic, we write $IMP \models t = t'$ or $IMP \models t \rightarrow t'$ when the equation $t = t'$ or the rule $t \rightarrow t'$, respectively, can be derived using rewriting logic deduction from the IMP specification. We also write $IMP \models t \rightarrow^* t'$ when the rule $t \rightarrow t'$ can be derived in zero or more steps. Recall that rewrite steps apply modulo equations in rewriting logic. To simplify writing, we may drop the “ $IMP \models$ ” when understood.

Computation $k \in K$ is **well-formed** iff it is equal (using equational reasoning within IMP’s semantics) to a well-formed IMP statement or expression. Computation k is **well-terminated** iff it is equal to the unit computation “ \cdot ” or to an integer value $i \in Int$.

Let IMP° be the algebraic specification $(\Sigma_{IMP}, \mathcal{E}_{IMP}^\circ)$ of **IMP configurations**, where $\mathcal{E}_{IMP}^\circ \subset \mathcal{E}_{IMP}$ contains all the configuration defining equations (those for bags, sequences, maps, etc., but not the computation or semantic equations). Let \mathcal{T} be the initial IMP° algebra, i.e., the Σ_{IMP} -algebra of \mathcal{E}° equational classes of ground terms (i.e., Σ_{IMP} -terms provably equal with equations in \mathcal{E}_{IMP}° are considered identical). Terms $\langle \langle k \rangle_k \langle \rho \rangle_{env} \rangle$ in \mathcal{T} of sort Cfg are called (**concrete**) **configurations** and we distinguish several types of them:

- Configurations $\langle \langle s \rangle_k \langle \cdot \rangle_{env} \rangle$ with $s \in S$, written more compactly $\llbracket s \rrbracket$, are called **initial configurations**;
- Configurations $\langle \langle k \rangle_k \langle \rho \rangle_{env} \rangle$ with k well-terminated are called **final configurations**;
- Configurations $\gamma \in \mathcal{T}$ which cannot be rewritten anymore are called **normal form configurations**;
- Normal form configurations which are not final are called **stuck (or junk, or core dump) configurations**;
- Configurations that cannot be rewritten infinitely are called **terminating configurations**.

For example, $\langle \langle x := x + y ; y := x - y \rangle_k \langle x \mapsto 7, y \mapsto 5 \rangle_{env} \rangle$ is a terminating configuration and, since bags are associative and commutative and environment maps are bags of pairs, it is identical to $\langle \langle y \mapsto 5, x \mapsto 7 \rangle_{env} \langle x := x + y ; y := x - y \rangle_k \rangle$. The configuration $\langle \langle x := 0 ; y := 1 \rangle_k \langle \cdot \rangle_{env} \rangle$ is initial, $\langle \langle \cdot \rangle_k \langle x \mapsto 0, y \mapsto 1 \rangle_{env} \rangle$ is final, and $\langle \langle x \leadsto \square + 1 \leadsto x := \square \rangle_k \langle y \mapsto 1 \rangle_{env} \rangle$ is stuck.

\mathbb{K} rules are schemas and, unlike in SOS [Plotkin 2004], they are *unconditional*; thus, no premises need to be proved in order to apply a rule. Note that the first rule for the conditional statement has the side condition “ $i \neq 0$ ”, but that acts as a simple filter for instances of the rule schema and not as a premise that needs to recursively invoke the rewriting procedure. Also, note that unlike in reduction semantics with evaluation contexts [Wright and Felleisen 1994], \mathbb{K} rewrites are *context insensitive*, in that its rules and equations can apply wherever they match; one needs to use structure to inhibit applications of undesired rules or equations, like we did for `while`. One natural question is whether the tasks in a computation structure

are indeed processed sequentially; the following result is crucial for most if not all subsequent results in the paper.

PROPOSITION 2. *Given $k \in E \cup S$ and $r \in K$, then $\langle\langle k \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \gamma$ for some final configuration γ iff there is some final configuration $\gamma' = \langle\langle k' \rangle_k \langle \rho' \rangle_{env}\rangle$ such that $\langle\langle k \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \gamma'$ and $\langle\langle k' \leadsto r \rangle_k \langle \rho' \rangle_{env}\rangle \rightarrow^* \gamma$; moreover, if that is the case then $k' = \rho(k)$ and $\rho' = \rho$ when $k \in E$, and $k' = \cdot$ when $k \in S$.*

Proof. The “if” part is easier; it states that if there are some final configurations $\gamma' = \langle\langle k' \rangle_k \langle \rho' \rangle_{env}\rangle$ and γ such that $\langle\langle k \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \gamma'$ and $\langle\langle k' \leadsto r \rangle_k \langle \rho' \rangle_{env}\rangle \rightarrow^* \gamma$, then $\langle\langle k \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \gamma$. Since each rewrite rule only modifies the computation at its top, we can repeat all the steps in the rewrite sequence $\langle\langle k \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle k' \rangle_k \langle \rho' \rangle_{env}\rangle$ starting with the term $\langle\langle k \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle$ and appending r to the computation of every other configuration that appears in the rewrite sequence, eventually obtaining a rewrite sequence $\langle\langle k \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle k' \leadsto r \rangle_k \langle \rho' \rangle_{env}\rangle$. Since we know that $\langle\langle k' \leadsto r \rangle_k \langle \rho' \rangle_{env}\rangle \rightarrow^* \gamma$, we conclude $\langle\langle k \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \gamma$.

We prove the “only if” part by structural induction on k .

Let us first consider the cases where $k \in E$. If $k \in Int$ then one can take $k' = k = \rho(k)$ and $\gamma' = \langle\langle k \rangle_k \langle \rho \rangle_{env}\rangle$, which verify the property. If $k \in PVar$ then there is only one possibility for the first step in the rewrite sequence of $\langle\langle k \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle$ to γ , namely to use the variable lookup rule, so one can take $k' = \rho(k)$ and $\rho' = \rho$, which verify the property. Suppose now that k is an expression of the form $e_1 \text{ op } e_2$. If $e_1, e_2 \in Int$ then take $k' = \rho(e_1 \text{ op } e_2) = e_1 \text{ op}_{Int} e_2$ and $\rho' = \rho$. If $e_1 \in Int$ and $e_2 \notin Int$ then, since $k = e_2 \leadsto e_1 \text{ op } \square$, by the induction hypothesis we get that $\langle\langle k \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \rho(e_2) \leadsto e_1 \text{ op } \square \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle$ and also that $\langle\langle \rho(e_2) \leadsto e_1 \text{ op } \square \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \gamma$. For the latter sequence, since $\rho(e_2) \leadsto e_1 \text{ op } \square = e_1 \text{ op } \rho(e_2)$, the only way for that rewriting sequence to take place is that is starts by rewriting $e_1 \text{ op } \rho(e_2)$ to $e_1 \text{ op}_{Int} \rho(e_2)$. Then take $k' = e_1 \text{ op}_{Int} \rho(e_2) = \rho(e_1 \text{ op } e_2)$ and $\rho' = \rho$ and note that the property holds. If $e_1 \notin Int$ then, since $k = e_1 \leadsto \square \text{ op } e_2$, by the induction hypothesis we get that $\langle\langle k \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \rho(e_1) \leadsto \square \text{ op } e_2 \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle$ and also that $\langle\langle \rho(e_1) \leadsto \square \text{ op } e_2 \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \gamma$. For the latter sequence, since $\rho(e_1) \leadsto \square \text{ op } e_2 = \rho(e_1) \text{ op } e_2 = e_2 \leadsto \rho(e_1) \text{ op } \square$, by the induction hypothesis, $\langle\langle e_2 \leadsto \rho(e_1) \text{ op } \square \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \rho(e_2) \leadsto \rho(e_1) \text{ op } \square \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \gamma$. For the latter sequence, since $\rho(e_2) \leadsto \rho(e_1) \text{ op } \square = \rho(e_1) \text{ op } \rho(e_2)$, the only way for that rewriting sequence to take place is that is starts by rewriting $\rho(e_1) \text{ op } \rho(e_2)$ to $\rho(e_1) \text{ op}_{Int} \rho(e_2)$. Then take $k' = \rho(e_1) \text{ op}_{Int} \rho(e_2) = \rho(e_1 \text{ op } e_2)$ and $\rho' = \rho$ and note that the property holds.

Let us now analyze the cases where $k \in S$. If $k = s_1; s_2 = s_1 \leadsto s_2$ then by the induction hypothesis there is some ρ_1 such that $\langle\langle s_1 \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \cdot \rangle_k \langle \rho_1 \rangle_{env}\rangle$ and $\langle\langle s_2 \leadsto r \rangle_k \langle \rho_1 \rangle_{env}\rangle \rightarrow^* \gamma$. The latter rewrite implies, by the induction hypothesis again, that there is some ρ_2 such that $\langle\langle s_2 \rangle_k \langle \rho_1 \rangle_{env}\rangle \rightarrow^* \langle\langle \cdot \rangle_k \langle \rho_2 \rangle_{env}\rangle$ and $\langle\langle r \rangle_k \langle \rho_2 \rangle_{env}\rangle \rightarrow^* \gamma$. Pick $k' = \cdot$ and $\rho' = \rho_2$ and note that the property verifies. If $k = (x := e) = e \leadsto (x := \square)$ then by the induction hypothesis $\langle\langle k \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle x := \rho(e) \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \gamma$; the only way to advance the rewriting of $\langle\langle x := \rho(e) \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle$ is to apply a variable update rule instance, so we obtain that $\langle\langle x := \rho(e) \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow \langle\langle r \rangle_k \langle \rho[\rho(e)/x] \rangle_{env}\rangle \rightarrow^* \gamma$. Then pick $k' = \cdot$ and $\rho' = \rho[\rho(e)/x]$ and note that the property holds. If $k = \text{if}(e) s_1 \text{ else } s_2 = e \leadsto \text{if}(\square) s_1 \text{ else } s_2$, then $\langle\langle k \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \text{if}(\rho(e)) s_1 \text{ else } s_2 \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \gamma$ by the induction hypothesis. Since $\rho(e) \in Int$, it is either 0 or different from 0. If $\rho(e) \neq 0$ then $\langle\langle k \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle s_1 \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \gamma$, so by the induction hypothesis there is some ρ' s.t. $\langle\langle s_1 \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle$ and $\langle\langle r \rangle_k \langle \rho' \rangle_{env}\rangle \rightarrow^* \gamma$. Pick $k' = \cdot$ and ρ' as above and note that the property holds. If $\rho(e) = 0$ then $\langle\langle k \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle s_2 \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \gamma$, so by the induction hypothesis there is some ρ' s.t. $\langle\langle s_2 \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^*$

(IMP statement rules)

$\frac{}{\{\varphi[e/x]\} x := e \{\varphi\}}$	(HL-ASGN)
$\frac{\{\varphi_1\} s_1 \{\varphi_2\}, \{\varphi_2\} s_2 \{\varphi_3\}}{\{\varphi_1\} s_1; s_2 \{\varphi_3\}}$	(HL-SEQ)
$\frac{\{\varphi \wedge (e \neq 0)\} s_1 \{\varphi'\}, \{\varphi \wedge (e = 0)\} s_2 \{\varphi'\}}{\{\varphi\} \text{if}(e) s_1 \text{ else } s_2 \{\varphi'\}}$	(HL-IF)
$\frac{\{\varphi \wedge (e \neq 0)\} s \{\varphi\}}{\{\varphi\} \text{while}(e) s \{\varphi \wedge (e = 0)\}}$	(HL-WHILE)
(Generic rule)	
$\frac{\models \psi \Rightarrow \varphi, \{\varphi\} s \{\varphi'\}, \models \varphi' \Rightarrow \psi'}{\{\psi\} s \{\psi'\}}$	(HL-CONSEQUENCE)

Figure 3. Hoare logic formal system for IMP

$\langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle$ and $\langle\langle r \rangle_k \langle \rho' \rangle_{env}\rangle \rightarrow^* \gamma$. Pick $k' = \cdot$ and ρ' as above and note that the property holds. Now suppose that $k = \text{while}(e) s$ and that $\langle\langle \text{while}(e) s \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^n \gamma$ for some $n > 0$. We prove the property by well-founded induction on n , for any ρ . Like above, it must be that $\langle\langle \text{while}(e) s \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \text{if}(\rho(e)) s; \text{while}(e) s \text{ else } \cdot \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^{n'} \gamma$ for some $n' \leq n$. If $\rho(e) \neq 0$ it can only be that the conditional rule is applied on the intermediate term above, then by the induction hypothesis there is some ρ'' such that $\langle\langle s \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \cdot \rangle_k \langle \rho'' \rangle_{env}\rangle$ and $\langle\langle \text{while}(e) s \leadsto r \rangle_k \langle \rho'' \rangle_{env}\rangle \rightarrow^{n''} \gamma$ for some $n'' < n'$ (at least one rewrite step has been consumed by the conditional). By the inner induction hypothesis (since $n'' < n$), there is some ρ' such that $\langle\langle \text{while}(e) s \rangle_k \langle \rho'' \rangle_{env}\rangle \rightarrow^* \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle$ and $\langle\langle r \rangle_k \langle \rho' \rangle_{env}\rangle \rightarrow^* \gamma$, which proves our property. If $\rho(e) = 0$ then the conditional rewrites to \cdot , so $\langle\langle \text{while}(e) s \leadsto r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle r \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \gamma$. Pick $k' = \cdot$ and $\rho' = \rho$ and note that the property holds. PROPOSITION 2

3.2 Hoare Logic Proof System for IMP

Figure 3 gives IMP an axiomatic semantics as a Hoare logic proof system deriving (partial) correctness triples of the form $\{\varphi\} s \{\varphi'\}$, where φ and φ' are FOL formulae called the *pre-condition* and the *post-condition*, respectively, and s is a statement; the intuition for $\{\varphi\} s \{\varphi'\}$ is that if φ holds before s is executed, then φ' holds whenever s terminates. We only consider partial correctness triples in this paper. In Hoare logic, program states are mappings (that we call environments) from variables to values, and are specified as FOL formulae — environment ρ “satisfies” φ iff $\rho \models \varphi$ in FOL. Because of this, program variables are regarded as logical variables in specifications; moreover, because of the rule (HL-ASGN) which may infuse program expression e into the pre-condition, specifications in Hoare logic actually extend program expressions, which is sometimes a source of confusion and technical inconvenience (since not all languages agree on the same syntax for expressions, since some expression constructs may have side effects or peculiar evaluation strategies, such as the “shortcut” and $\&\&$ in C, etc.).

Let us formally define the underlying FOL. Its signature is the subsignature of IMP in Figure 2 keeping the expressions E and their subsorts (i.e., excluding only S and the statement constructs). Its universe of variables, Var , is $PVar$. Assume a background theory (i.e., a set of formulae that are by default conjuncted with any other formula) that can prove $i_1 \text{ op } i_2 = i_1 \text{ op}_{Int} i_2$ for any $i_1, i_2 \in Int$. To ease writing, expressions e are allowed as formulae as syntactic sugar for $\neg(e = 0)$. Here are two examples of correctness triples:

$$\begin{aligned} \{x > 0 \wedge z = \text{old}_z\} \quad z := x + z \quad \{z > \text{old}_z\} \\ \{\exists z. (x = 2 * z + 1)\} \quad x := x * x + x \quad \{\exists z. (x = 2 * z)\} \end{aligned}$$

The first sequent above says that the new value of z is larger after the assignment statement than the old value, and the second says that if x is odd before the assignment then it is even after.

We now fix a FOL model, say Int , whose carrier corresponding to the sort E of expressions is the integer numbers. Finite-domain valuations $\rho : PVar \rightarrow Int$ in this model then correspond to environment mappings as defined in Figure 2. We can now state the soundness theorem of the Hoare logic formal system for IMP in Figure 3 w.r.t. its \mathbb{K} executable definition in Figure 2 (as mentioned, here we only consider partial correctness). The role of THEOREM 3 is twofold: on the one hand it shows how \mathbb{K} definitions relate to Hoare logic formal systems, and on the other hand it prepares the ground for the soundness of matching logic (THEOREM 8).

THEOREM 3. (Soundness of Hoare logic w.r.t. \mathbb{K} for IMP) *If $\{\varphi\} s \{\varphi'\}$ is derivable, then for any environment $\rho \in \text{Map}^{--}[PVar, Int]$ with $\rho \models \varphi$, if $\text{IMP} \models \langle\langle s \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle$ then $\rho' \models \varphi'$.*

Proof. By structural induction on the derivation of $\{\varphi\} s \{\varphi'\}$ using the proof system in Figure 3. We next consider each rule.

$$\frac{}{\{\varphi[e/x]\} x := e \{\varphi\}} \quad (\text{HL-ASGN})$$

Let $\rho \in \text{Map}^{--}[PVar, Int]$ be such that $\rho \models \varphi[e/x]$, and suppose that $\langle\langle x := e \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle$. We claim that $\rho' = \rho[\rho(e)/x]$. Indeed, since $\langle\langle x := e \rangle_k \langle \rho \rangle_{env}\rangle = \langle\langle e \leadsto x := \square \rangle_k \langle \rho \rangle_{env}\rangle$, by Proposition 2 we get $\langle\langle e \leadsto x := \square \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \rho(e) \leadsto x := \square \rangle_k \langle \rho \rangle_{env}\rangle$, and so the original rewrite sequence can only be of the form $\langle\langle x := e \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle x := \rho(e) \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle$, which implies that $\rho' = \rho[\rho(e)/x]$. Next, since we assumed that $\rho \models \varphi[e/x]$, it follows that $\rho' \models \varphi$ (see fact listed at the end of Section 2).

$$\frac{\{\varphi_1\} s_1 \{\varphi_2\}, \{\varphi_2\} s_2 \{\varphi_3\}}{\{\varphi_1\} s_1 ; s_2 \{\varphi_3\}} \quad (\text{HL-SEQ})$$

Let $\rho_1 \in \text{Map}^{--}[PVar, Int]$ such that $\rho_1 \models \varphi_1$, and suppose that $\langle\langle s_1 ; s_2 \rangle_k \langle \rho_1 \rangle_{env}\rangle \rightarrow^* \langle\langle \cdot \rangle_k \langle \rho_3 \rangle_{env}\rangle$. Since $\langle\langle s_1 ; s_2 \rangle_k \langle \rho_1 \rangle_{env}\rangle = \langle\langle s_1 \leadsto s_2 \rangle_k \langle \rho_1 \rangle_{env}\rangle$, by Proposition 2 we get that there is some $\rho_2 \in \text{Map}^{--}[PVar, Int]$ such that $\langle\langle s_1 \rangle_k \langle \rho_1 \rangle_{env}\rangle \rightarrow^* \langle\langle \cdot \rangle_k \langle \rho_2 \rangle_{env}\rangle$ and $\langle\langle s_2 \rangle_k \langle \rho_2 \rangle_{env}\rangle \rightarrow^* \langle\langle \cdot \rangle_k \langle \rho_3 \rangle_{env}\rangle$. By the induction hypothesis for the derivation of $\{\varphi_1\} s_1 \{\varphi_2\}$ we get $\rho_2 \models \varphi_2$, and then $\rho_3 \models \varphi_3$ by the induction hypothesis for the derivation of $\{\varphi_2\} s_2 \{\varphi_3\}$.

$$\frac{\{\varphi \wedge (e \neq 0)\} s_1 \{\varphi'\}, \{\varphi \wedge (e = 0)\} s_2 \{\varphi'\}}{\{\varphi\} \text{if } (e) s_1 \text{ else } s_2 \{\varphi'\}} \quad (\text{HL-IF})$$

Let $\rho \in \text{Map}^{--}[PVar, Int]$ be such that $\rho \models \varphi$, and suppose that $\langle\langle \text{if } (e) s_1 \text{ else } s_2 \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle$. Since the equality $\langle\langle \text{if } (e) s_1 \text{ else } s_2 \rangle_k \langle \rho \rangle_{env}\rangle = \langle\langle e \leadsto \text{if } (\square) s_1 \text{ else } s_2 \rangle_k \langle \rho \rangle_{env}\rangle$ holds, by Proposition 2 we get that $\langle\langle \text{if } (e) s_1 \text{ else } s_2 \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \text{if } (\rho(e)) s_1 \text{ else } s_2 \rangle_k \langle \rho \rangle_{env}\rangle$. We now distinguish two cases: if $\rho(e) \neq 0$ then $\langle\langle \text{if } (\rho(e)) s_1 \text{ else } s_2 \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow \langle\langle s_1 \rangle_k \langle \rho \rangle_{env}\rangle$; if $\rho(e) = 0$ then $\langle\langle \text{if } (\rho(e)) s_1 \text{ else } s_2 \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow \langle\langle s_2 \rangle_k \langle \rho \rangle_{env}\rangle$. The first case implies that $\rho \models \varphi \wedge (e \neq 0)$ and that $\langle\langle s_1 \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle$, so by the induction hypothesis for the derivation of $\{\varphi \wedge (e \neq 0)\} s_1 \{\varphi'\}$ it follows that $\rho' \models \varphi'$. The second case implies that $\rho \models \varphi \wedge (e = 0)$ and that $\langle\langle s_2 \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle$, so by the induction hypothesis for the derivation of $\{\varphi \wedge (e = 0)\} s_2 \{\varphi'\}$ it follows that $\rho' \models \varphi'$.

$$\frac{\{\varphi \wedge (e \neq 0)\} s \{\varphi\}}{\{\varphi\} \text{while } (e) s \{\varphi \wedge (e = 0)\}} \quad (\text{HL-WHILE})$$

(IMP statement rules)

$$\frac{C[e] \equiv v}{\langle C \rangle x := e \langle C[x \leftarrow v] \rangle} \quad (\text{ML-ASGN})$$

$$\frac{\langle C_1 \rangle s_1 \langle C_2 \rangle, \langle C_2 \rangle s_2 \langle C_3 \rangle}{\langle C_1 \rangle s_1 ; s_2 \langle C_3 \rangle} \quad (\text{ML-SEQ})$$

$$\frac{C[e] \equiv v, \langle C \wedge (v \neq 0) \rangle s_1 \langle C' \rangle, \langle C \wedge (v = 0) \rangle s_2 \langle C' \rangle}{\langle C \rangle \text{if } (e) s_1 \text{ else } s_2 \langle C' \rangle} \quad (\text{ML-IF})$$

$$\frac{C[e] \equiv v, \langle C \wedge (v \neq 0) \rangle s \langle C' \rangle}{\langle C \rangle \text{while } (e) s \langle C \wedge (v = 0) \rangle} \quad (\text{ML-WHILE})$$

(Generic rule)

$$\frac{\models \Gamma \Rightarrow \Gamma_1, \Gamma_1 \Downarrow \Gamma'_1, \models \Gamma'_1 \Rightarrow \Gamma'}{\Gamma \Downarrow \Gamma'} \quad (\text{ML-CONSEQUENCE})$$

(only statement patterns)

(IMP expression rules)

$$\frac{}{C[i] \equiv i} \quad (\text{ML-INT})$$

$$\frac{}{C \langle x \mapsto v, \rho \rangle_{env} [x] \equiv v} \quad (\text{ML-LOOKUP})$$

$$\frac{C[e_1] \equiv v_1, C[e_2] \equiv v_2}{C[e_1 \text{ op } e_2] \equiv v_1 \text{ op}_{Int} v_2} \quad (\text{ML-OP})$$

Figure 4. Matching logic formal system for IMP

Let $\rho \in \text{Map}^{--}[PVar, Int]$ be such that $\rho \models \varphi$, and suppose that $\langle\langle \text{while } (e) s \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^n \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle$ for some $n > 0$. We prove by well-founded induction on n that $\rho' \models \varphi \wedge (e = 0)$. By Proposition 2 it follows that $\langle\langle \text{while } (e) s \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \text{if } (\rho(e)) s ; \text{while } (e) s \text{ else } \cdot \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^{n'} \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle$, where $n' \leq n$. We again distinguish two cases, like in the case of the conditional: if $\rho(e) \neq 0$ then $\langle\langle \text{if } (\rho(e)) s ; \text{while } (e) s \text{ else } \cdot \rangle_k \langle \rho \rangle_{env}\rangle$ rewrites in one step to $\langle\langle s ; \text{while } (e) s \rangle_k \langle \rho \rangle_{env}\rangle$, and if $\rho(e) = 0$ then it rewrites in one step to $\langle\langle \cdot \rangle_k \langle \rho \rangle_{env}\rangle$. The second case is simpler: it implies that $\rho \models \varphi \wedge (e = 0)$ and that $\rho' = \rho$, so $\rho' \models \varphi \wedge (e = 0)$. The first case implies that $\rho \models \varphi \wedge (e \neq 0)$ and that $\langle\langle s ; \text{while } (e) s \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^{n'-1} \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle$. By Proposition 2 again it follows that there is some $\rho'' \in \text{Map}^{--}[PVar, Int]$ such that $\langle\langle s \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \cdot \rangle_k \langle \rho'' \rangle_{env}\rangle$ and $\langle\langle \text{while } (e) s \rangle_k \langle \rho'' \rangle_{env}\rangle \rightarrow^{n''} \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle$ for some $n'' < n'$. By the induction hypothesis for the derivation of $\{\varphi \wedge (e \neq 0)\} s \{\varphi\}$ it follows that $\rho'' \models \varphi$. By the well-founded induction hypothesis for $n'' < n$ (and ρ''), it follows that $\rho' \models \varphi \wedge (e = 0)$.

$$\frac{\models \psi \Rightarrow \varphi, \{\varphi\} s \{\varphi'\}, \models \varphi' \Rightarrow \psi'}{\{\psi\} s \{\psi'\}} \quad (\text{HL-CONSEQUENCE})$$

Let $\rho \in \text{Map}^{--}[PVar, Int]$ be such that $\rho \models \psi$, and suppose that $\langle\langle s \rangle_k \langle \rho \rangle_{env}\rangle \rightarrow^* \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle$. Since $\models \psi \Rightarrow \varphi$, it follows that $\rho \models \varphi$, so by the induction hypothesis we get that $\rho' \models \varphi'$, and since $\models \varphi' \Rightarrow \psi'$ we finally get $\rho' \models \psi'$. Q.E.D. THEOREM 3

4. Matching Logic Definition of IMP

In this section we first introduce some generic matching logic notions and notations, and then give a matching logic axiomatic semantics for IMP and show it sound w.r.t. the \mathbb{K} semantics in Section 3. Then we show that for the simple IMP language, a restricted use of this matching logic proof system is equivalent to the Hoare logic proof system of IMP in Section 3.2.

4.1 General Matching Logic Notions and Notations

A matching logic formal system for a language \mathcal{L} can be best given when one already has a \mathbb{K} semantics for \mathcal{L} . A technique is suggested in [Roşu and Schulte 2009] to relatively mechanically generate a matching logic formal system from a \mathbb{K} semantics, and in this paper we show how a program verifier can be derived from the \mathbb{K} semantics of a language. Additionally, as shown in [Meseguer and Roşu 2007] among other papers, a \mathbb{K} definition of a language also yields an interpreter for the language as well as an explicit state model checker essentially for free when executed in Maude. Thus, there are multiple benefits of giving a language a \mathbb{K} semantics. However, supposing that one wants to exclusively define a matching logic proof system for a language and is not interested in the other benefits of a \mathbb{K} semantics, all what is really needed is an appropriate definition of configurations; having a fully executable and tested \mathbb{K} semantic definition of \mathcal{L} gives confidence in the appropriateness of its configurations, but it is not necessary.

Let $\mathcal{L}^\circ = (\Sigma_{\mathcal{L}}, \mathcal{E}_{\mathcal{L}}^\circ)$ be the algebraic specification defining the configurations of some language \mathcal{L} ; Section 3 (Definition 1) discusses IMP° as an example of a two-subcell configuration (a computation and an environment) and one other cell is added in Section 6 (a heap), but one can have arbitrarily complex configurations in \mathcal{L}° . Let Cfg be the sort of configurations in \mathcal{L}° . A first major distinction between matching logic and Hoare logic is that in matching logic *program variables are syntactic constants, not logical variables*. In other words, one cannot quantify over program variables and they are neither free nor bound in formulae. Instead, let Var be a sortwise infinite set of logical, or semantical variables, together with a distinguished variable named “ \circ ” of sort Cfg which is going to serve as a placeholder for the configuration in specifications.

DEFINITION 4. *Matching logic specifications, called **configuration patterns** or just **patterns**, are $\text{FOL}_=$ formulae $\exists X.(\circ = c \wedge \varphi)$, where:*

- $X \subset \text{Var}$ is the set of (**pattern**) **bound variables**; the variables in c and the free variables in φ which are not in X are the (**pattern**) **free variables**; special variable “ \circ ” is considered neither free nor bound, and can only appear once in the pattern, as shown;
- c is the **pattern structure** and is a term of sort Cfg that may contain logical variables in Var (bound or not to the pattern); it may (and typically will) also contain program variables in PVar , but recall that those have no logical meaning;
- φ is the (**pattern**) **constraints**, an arbitrary $\text{FOL}_=$ formula.

We let Γ, Γ', \dots range over patterns. For example, the IMP pattern $\exists z.(\circ = \langle \langle x := y/x \rangle_k (x \mapsto x, y \mapsto x *_{\text{Int}} z, \rho)_{\text{env}} \rangle \wedge x \neq 0)$ specifies configurations whose computation is “ $x := y/x$ ” and in which the value x held by x is different from 0 and divides the value held by y ; variables x and ρ are free, meaning that they are expected to be bound by the proof context (like in Hoare logic, such free variables in specifications act as parameters to the entire proof).

Let us fix as underlying model $\mathcal{T}_{\mathcal{L}^\circ}$, written more compactly just \mathcal{T} , the initial model of \mathcal{L}° ; \mathcal{T} is obtained by factoring the $\Sigma_{\mathcal{L}}$ -algebra of ground terms by all the equational properties in $\mathcal{E}_{\mathcal{L}}^\circ$, so \mathcal{T} is the model of concrete (i.e., non-symbolic) values and configurations — Section 3 (Definition 1) defines the corresponding initial model for IMP° . Let Var° be the set $\text{Var} \cup \{\circ\}$ of variables Var extended with the special variable \circ of sort Cfg . Valuations $\text{Var}^\circ \rightarrow \mathcal{T}$ then consist of a concrete configuration γ corresponding to \circ and of a map $\tau : \text{Var} \rightarrow \mathcal{T}$; we write such valuations using a pairing notation, namely $(\gamma, \tau) : \text{Var}^\circ \rightarrow \mathcal{T}$.

We are now ready to introduce the concept of *pattern matching* that inspired the name of our axiomatic semantic approach.

DEFINITION 5. *Configuration γ **matches** pattern $\exists X.(\circ = c \wedge \varphi)$, say Γ , iff there is some $\tau : \text{Var} \rightarrow \mathcal{T}$ such that $(\gamma, \tau) \models \Gamma$.*

Let us elaborate on the meaning of $(\gamma, \tau) \models \exists X.(\circ = c \wedge \varphi)$. Since the special variable \circ appears only once in the pattern and since the syntax of configurations is included in the syntax of our $\text{FOL}_=$, it is equivalent to $\tau \models \exists X.(\gamma = c \wedge \varphi)$, which is further equivalent to saying that there exists some $\theta_\tau : \text{Var} \rightarrow \mathcal{T}$ with $\theta_\tau \upharpoonright_{\text{Var} \setminus X} = \tau \upharpoonright_{\text{Var} \setminus X}$ such that $\gamma = \theta_\tau(c) \wedge \theta_\tau(\varphi)$ holds. Note that $\gamma = \theta_\tau(c) \wedge \theta_\tau(\varphi)$ is a closed formula, because θ_τ substitutes a ground term for each variable; in particular, if v is a term (possibly containing semantic variables) of sort Int appearing in c , then $\theta_\tau(v)$ is an integer number. Therefore, this is equivalent to saying that $\gamma = \theta_\tau(c)$ and $\theta_\tau \models \varphi$.

We next introduce an important syntactic sugar notation for the common case when configurations are bags of cells, that is, when “ $\text{Cfg} ::= (\text{Bag} \rightarrow [\text{CfgItem}])$ ”. If that is the case, then we let C, C', \dots , range over configuration item bag terms, possibly with variables.

NOTATION 6. *If \mathcal{L} ’s configuration syntax is of the form “ $\text{Cfg} ::= (\text{Bag} \rightarrow [\text{CfgItem}])$ ” and if $\langle C \rangle$ is a Cfg -term, then we may write $\langle C \rangle_{\text{bnd}} \langle \varphi \rangle_{\text{form}}$ instead of $\exists X.(\circ = \langle C \rangle \wedge \varphi)$. We continue to call the structures $\langle C \rangle_{\text{bnd}} \langle \varphi \rangle_{\text{form}}$ **patterns**.*

In other words, we include the pattern bound variables X and constraints φ as subcells in the top cell, thus regarding them as additional ingredients in the pattern configuration structure. The rationale for this notation is twofold: on the one hand it eases the writing and reading of matching logic proof systems by allowing meta-variables C, C' to also range over the two additional subcells when not important in a given context, and on the other hand it prepares the ground for our technique to derive matching logic provers from \mathbb{K} executable semantics. This is because the pattern bound variables and constraints are effectively added as new cells into configurations, so most of the equations and rules in the \mathbb{K} semantics can be borrowed and used *unchanged* in the prover!

A matching logic axiomatic semantics to a programming language is given as a proof system for deriving special sequents called *correctness pairs*, which relate configurations before the execution of a fragment of program to configurations after its execution:

DEFINITION 7. *A matching logic (**partial**) **correctness pair** consists of two patterns Γ and Γ' , written $\Gamma \Downarrow \Gamma'$. For analogy to Hoare logic, we call Γ a **pre-condition pattern** and Γ' a **post-condition pattern**.*

Since the code is included as part of the pre-condition pattern Γ , we do not mention it as part of the sequent as in Hoare triples; to ease reading, we however shortly introduce some notational conventions that will highlight the code. Section 4.2 gives a matching logic proof system for IMP with configurations consisting of a computation and an environment, and Section 6 extends it for HIMP , whose configurations add a heap to IMP ’s configurations.

4.2 Matching Logic Proof System for IMP

Figure 4 gives a matching logic proof system for IMP . To make it resemble the more familiar Hoare logic proof system of IMP , we adopted the following additional notational conventions:

$$\begin{aligned} \langle C \rangle_{\text{env}}[\mathbf{x} \leftarrow v] & \text{ instead of } C \langle \rho[v/\mathbf{x}] \rangle_{\text{env}} \\ \langle C \rangle_{\text{form}} \wedge \varphi' & \text{ instead of } C \langle \varphi \wedge \varphi' \rangle_{\text{form}} \\ C[\mathbf{e}] \equiv v & \text{ instead of } \langle \langle \mathbf{e} \rangle_k C \rangle \Downarrow \langle \langle v \rangle_k C \rangle \\ \langle C \rangle \mathbf{s} \langle C' \rangle & \text{ instead of } \langle C \rangle_{\mathbf{s}} \langle C' \rangle \Downarrow \langle C' \rangle_{\mathbf{s}} \end{aligned}$$

The meta-variables C, C' above range over appropriate configuration item bag terms so that the resulting patterns are well-formed for IMP° (see Definition 1). The desugared rule for while is then:

$$\frac{\begin{array}{c} \exists X.(\circ = \langle \langle \mathbf{e} \rangle_k C \rangle \wedge \varphi) \Downarrow \exists X.(\circ = \langle \langle v \rangle_k C \rangle \wedge \varphi), \\ \exists X.(\circ = \langle \langle \mathbf{s} \rangle_k C \rangle \wedge \varphi \wedge (v \neq 0)) \Downarrow \exists X.(\circ = \langle \langle \cdot \rangle_k C \rangle \wedge \varphi) \end{array}}{\exists X.(\circ = \langle \langle \text{while } (\mathbf{e}) \mathbf{s} \rangle_k C \rangle \wedge \varphi) \Downarrow \exists X.(\circ = \langle \langle \cdot \rangle_k C \rangle \wedge \varphi \wedge (v = 0))}$$

```

 $\langle\langle p \mapsto p, s \mapsto s, n \mapsto n, \rho \rangle_{env} c \langle p \geq 0 \rangle_{form} \langle \cdot \rangle_{bnd} \rangle$ 
 $s = 0; \quad n = 1;$ 
 $\langle\langle p \mapsto p, s \mapsto 0, n \mapsto 1, \rho \rangle_{env} c \langle p \geq 0 \rangle_{form} \langle \cdot \rangle_{bnd} \rangle$ 
 $\langle\langle p \mapsto p, s \mapsto n(n-1)/2, n \mapsto n, \rho \rangle_{env} c \langle p \geq 0 \wedge n \leq p+1 \rangle_{form} \langle n \rangle_{bnd} \rangle$ 
 $\text{while}(n \neq p + 1) \{$ 
 $\langle\langle p \mapsto p, s \mapsto n(n-1)/2, n \mapsto n, \rho \rangle_{env} c \langle p \geq 0 \wedge n < p+1 \rangle_{form} \langle n \rangle_{bnd} \rangle$ 
 $\quad s = s + n; \quad n = n + 1$ 
 $\langle\langle p \mapsto p, s \mapsto n(n+1)/2, n \mapsto n+1, \rho \rangle_{env} c \langle p \geq 0 \wedge n < p+1 \rangle_{form} \langle n \rangle_{bnd} \rangle$ 
 $\langle\langle p \mapsto p, s \mapsto n(n-1)/2, n \mapsto n, \rho \rangle_{env} c \langle p \geq 0 \wedge n \leq p+1 \rangle_{form} \langle n \rangle_{bnd} \rangle$ 
 $\}$ 
 $\langle\langle p \mapsto p, s \mapsto n(n-1)/2, n \mapsto n, \rho \rangle_{env} c \langle p \geq 0 \wedge n = p+1 \rangle_{form} \langle n \rangle_{bnd} \rangle$ 
 $\langle\langle p \mapsto p, s \mapsto p(p+1)/2, n \mapsto p+1, \rho \rangle_{env} c \langle p \geq 0 \rangle_{form} \langle \cdot \rangle_{bnd} \rangle$ 

```

Figure 5. Matching logic proof of the sum of first p numbers.

In the case of IMP, C above ranges over one-element configuration item bags, namely $\langle \rho \rangle_{env}$. However, in the case of HIMP, C will include an additional heap cell. Using generic meta-variables like C above instead of more concrete configuration item bag terms is key to the modularity of our matching logic definitions and proofs. Indeed, to add heaps to IMP in Section 6 we only add new rules for the new language constructs (none of the rules in Figure 4 changes).

Figure 5 shows a matching logic proof, following a compact and intuitive notation, also used in informal Hoare logic proofs. The grayed text is the code, the rest are proof annotations. A sequence $\langle C \rangle s \langle C' \rangle$ means that a proof for the corresponding correctness pair can be derived, and two consecutive patterns means that the former implies the latter, so an implicit use of (ML-CONSEQUENCE) is assumed. Note that p, s, n, ρ and c are free, so they act as parameters for the proof. Since s and n do not occur in the post-pattern anymore, it means that the program calculates the sum of numbers up to p no matter what the initial values of s and n were. Also, note that the variables ρ and c appear unaltered in the post-pattern, meaning that nothing else changes in the environment and in the configuration pattern; in the case of IMP the c is always empty (“.”), but by placing it there we can transport the proof as is to HIMP, in which case c says that the heap remains unchanged.

The variables ρ and c above act as environment and configuration frames. In matching logic, one can have a frame corresponding to any cell. We refrain from adding framing rules for these because one can achieve the same result methodologically, by always using a free variable matching “the rest” of the cell one is interested to frame. This works for the heap the same way. We refer the reader to [Roşu and Schulte 2009] for more on framing in matching logic.

The next result shows that the matching logic proof system for IMP in Figure 4 is sound w.r.t. the \mathbb{K} executable semantics in Figure 2. For technical reasons, we assume that the original IMP program (embedded in the pattern Γ below) is ground (i.e., it does not contain variables in Var); this is not a limitation, because the program variables in $PVar$ are treated as constants in matching logic. If one wants programs to be parametric, one can do it by using free variables in the environment, as in Figure 5.

THEOREM 8. (Soundness of matching logic w.r.t. \mathbb{K} for IMP) *If $\Gamma \Downarrow \Gamma'$ is derivable, then for any $(\gamma, \tau) : Var^\circ \rightarrow \mathcal{T}$ with $(\gamma, \tau) \models \Gamma$, if $\text{IMP} \models \gamma \rightarrow^* \gamma'$ with γ' final then $(\gamma', \tau) \models \Gamma'$.*

Proof. For simplicity of writing, we do the proof specifically to the current definition and implicit configuration structure of IMP. In other words, we consider that the top-level configurations only have a computation cell $\langle \dots \rangle_k$ and an environment cell $\langle \dots \rangle_{env}$. However, we never use the fact that configurations have only these two items, so our soundness proof below works for more complex configurations, too, like those in Section 6 for the HIMP language.

We first prove the soundness of the rules deriving “expression evaluation” sequents of the form $C[e] \equiv v$, that is, in desugared

form, $\exists X.(\circ = \langle \langle e \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X.(\circ = \langle \langle v \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi)$, where e and v are terms of sorts E and Int , respectively (note that v may contain semantic variables, bound or not). We prove a stronger result, namely that the resulting θ_τ ’s (see discussion preceding this theorem) can be chosen to be the same; specifically, we prove

LEMMA 9. *If $\exists X.(\circ = \langle \langle e \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X.(\circ = \langle \langle v \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi)$ is derivable and if $(\gamma, \tau) \models \exists X.(\circ = \langle \langle e \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi)$, which is equivalent to $\gamma = \langle \langle e \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$ for some $\theta_\tau : Var \rightarrow \mathcal{T}$ with $\theta_\tau \upharpoonright_{Var \setminus X} = \tau \upharpoonright_{Var \setminus X}$ and $\theta_\tau \models \varphi$, and if $\text{IMP} \models \gamma \rightarrow^* \gamma'$ with γ' final, then $\gamma' = \langle \langle \theta_\tau(v) \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$ (note that $\theta_\tau(v)$ is an integer); this implies, in particular, that $(\gamma', \tau) \models \exists X.(\circ = \langle \langle v \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi)$.*

Proof. The proof of LEMMA 9 goes by induction on the length of the derivation of $C[e] \equiv v$ using the proof system in Figure 4. For each of the rules deriving a sequent $C[e] \equiv v$ (desugared as above), consider $(\gamma, \tau) \models \exists X.(\circ = \langle \langle e \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi)$, that is, $\gamma = \langle \langle e \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$ for some $\theta_\tau : Var \rightarrow \mathcal{T}$ such that $\theta_\tau \upharpoonright_{Var \setminus X} = \tau \upharpoonright_{Var \setminus X}$ and $\theta_\tau \models \varphi$, and also that $\text{IMP} \models \gamma \rightarrow^* \gamma'$ with γ' final.

(ML-INT)

$$\frac{}{\exists X.(\circ = \langle \langle i \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X.(\circ = \langle \langle i \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi)}$$

In this case $\gamma = \langle \langle i \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$ is already final, so $\gamma' = \gamma = \langle \langle \theta_\tau(i) \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$ (note that $\theta_\tau(i) = i$).

(ML-LOOKUP)

$$\frac{}{\exists X.(\circ = \langle \langle x \rangle_k \langle x \mapsto v, \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X.(\circ = \langle \langle v \rangle_k \langle x \mapsto v, \rho \rangle_{env} \rangle \wedge \varphi)}$$

In this case $\gamma = \langle \langle x \rangle_k \langle x \mapsto \theta_\tau(v), \theta_\tau(\rho) \rangle_{env} \rangle$. The only way to rewrite γ in the \mathbb{K} semantics in Figure 2 is by a lookup rule (getting a final configuration), so $\gamma' = \langle \langle \theta_\tau(v) \rangle_k \langle \theta_\tau(x \mapsto v, \rho) \rangle_{env} \rangle$.

(ML-OP)

$$\frac{\begin{array}{l} \exists X.(\circ = \langle \langle e_1 \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X.(\circ = \langle \langle v_1 \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi), \\ \exists X.(\circ = \langle \langle e_2 \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X.(\circ = \langle \langle v_2 \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \end{array}}{\exists X.(\circ = \langle \langle e_1 \text{ op } e_2 \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X.(\circ = \langle \langle v_1 \text{ op }_{Int} v_2 \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi)}$$

In this case $\gamma = \langle \langle e_1 \text{ op } e_2 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle = \langle \langle e_1 \sim \square \text{ op } e_2 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$, so by Proposition 2 there is some integer i_1 s.t. $\langle \langle e_1 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle \rightarrow^* \langle \langle i_1 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$ and $\langle \langle i_1 \sim \square \text{ op } e_2 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$, which is equal to $\langle \langle i_1 \text{ op } e_2 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$, equal to $\langle \langle e_2 \sim i_1 \text{ op } \square \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$, rewrites to γ' . Using Proposition 2 again, it follows that there is some integer i_2 s.t. $\langle \langle e_2 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle \rightarrow^* \langle \langle i_2 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$ and $\langle \langle i_2 \sim i_1 \text{ op } \square \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle = \langle \langle i_1 \text{ op } i_2 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$ rewrites to γ' . It is obvious then that $\gamma' = \langle \langle i_1 \text{ op }_{Int} i_2 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$. By the induction hypothesis applied for $\gamma_1 =_{\text{def}} \langle \langle e_1 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$ and $\gamma_2 =_{\text{def}} \langle \langle e_2 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$, it follows that $i_1 = \theta_\tau(v_1)$ and $i_2 = \theta_\tau(v_2)$. Therefore, $\gamma' = \langle \langle \theta_\tau(v_1 \text{ op }_{Int} v_2) \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$. Q.E.D. LEMMA 9

LEMMA 9 implies the soundness of the expression rules. We now return to the proof of THEOREM 8 and prove the soundness of the statement rules. As expected, the proof also goes by induction, on the length of the derivation of the statement sequent $\langle C \rangle s \langle C' \rangle$ appearing in the conclusion of each rule of the proof system in Figure 4. As before, we consider these rules in desugared form. For each of the rules deriving a sequent of the form $\langle C \rangle s \langle C' \rangle$, desugared to $\exists X.(\circ = \langle \langle s \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X'.(\circ = \langle \langle \cdot \rangle_k \langle \rho' \rangle_{env} \rangle \wedge \varphi')$, consider that $(\gamma, \tau) \models \exists X.(\circ = \langle \langle s \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi)$, that is, that $\gamma = \langle \langle s \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$ for some $\theta_\tau : Var \rightarrow \mathcal{T}$ s.t. $\theta_\tau \upharpoonright_{Var \setminus X} = \tau \upharpoonright_{Var \setminus X}$ and $\theta_\tau \models \varphi$, and also consider that $\text{IMP} \models \gamma \rightarrow^* \gamma'$ with γ' final.

(ML-ASGN)

$$\frac{\exists X.(o = \langle \langle e \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X.(o = \langle \langle v \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi)}{\exists X.(o = \langle \langle x := e \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X.(o = \langle \langle \cdot \rangle_k \langle \rho[v/x] \rangle_{env} \rangle \wedge \varphi)}$$

In this case $\gamma = \langle \langle x := e \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle = \langle \langle e \rightsquigarrow x := \square \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$, so by Proposition 2 there is some integer i s.t. $\langle \langle e \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle \rightarrow^* \langle \langle i \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$ and $\langle \langle x := i \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle \rightarrow^* \gamma'$, so γ' can only be $\langle \langle \cdot \rangle_k \langle \theta_\tau(\rho)[i/x] \rangle_{env} \rangle$. By LEMMA 9, we get that $i = \theta_\tau(v)$. Then $\gamma' = \langle \langle \cdot \rangle_k \langle \theta_\tau(\rho[v/x]) \rangle_{env} \rangle$, so $(\gamma', \tau) \models \exists X.(o = \langle \langle \cdot \rangle_k \langle \rho[v/x] \rangle_{env} \rangle \wedge \varphi)$.

(ML-SEQ)

$$\frac{\begin{array}{l} \exists X_1.(o = \langle \langle s_1 \rangle_k \langle \rho_1 \rangle_{env} \rangle \wedge \varphi_1) \Downarrow \exists X_2.(o = \langle \langle \cdot \rangle_k \langle \rho_2 \rangle_{env} \rangle \wedge \varphi_2), \\ \exists X_2.(o = \langle \langle s_2 \rangle_k \langle \rho_2 \rangle_{env} \rangle \wedge \varphi_2) \Downarrow \exists X_3.(o = \langle \langle \cdot \rangle_k \langle \rho_3 \rangle_{env} \rangle \wedge \varphi_3) \end{array}}{\exists X_1.(o = \langle \langle s_1; s_2 \rangle_k \langle \rho_1 \rangle_{env} \rangle \wedge \varphi_1) \Downarrow \exists X_3.(o = \langle \langle \cdot \rangle_k \langle \rho_3 \rangle_{env} \rangle \wedge \varphi_3)}$$

In this case, $\gamma = \langle \langle s_1; s_2 \rangle_k \langle \theta_\tau^1(\rho_1) \rangle_{env} \rangle = \langle \langle s_1 \rightsquigarrow s_2 \rangle_k \langle \theta_\tau^1(\rho_1) \rangle_{env} \rangle$, where, for uniformity in writing, we replaced θ_τ by θ_τ^1 . By Proposition 2 and the induction hypothesis, we get $\langle \langle s_1 \rangle_k \langle \theta_\tau^1(\rho_1) \rangle_{env} \rangle \rightarrow^* \langle \langle \cdot \rangle_k \langle \theta_\tau^2(\rho_2) \rangle_{env} \rangle$ and $\langle \langle s_2 \rangle_k \langle \theta_\tau^2(\rho_2) \rangle_{env} \rangle \rightarrow^* \gamma'$ for some $\theta_\tau^2 : Var \rightarrow \mathcal{T}$ with $\theta_\tau^2 \upharpoonright_{Var \setminus X_2} = \tau \upharpoonright_{Var \setminus X_2}$ and $\theta_\tau^2 \models \varphi_2$. By the induction hypothesis again we get $(\gamma', \tau) \models \exists X_3.(o = \langle \langle \cdot \rangle_k \langle \rho_3 \rangle_{env} \rangle \wedge \varphi_3)$.

(ML-IF)

$$\frac{\begin{array}{l} \exists X.(o = \langle \langle e \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X.(o = \langle \langle v \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi), \\ \exists X.(o = \langle \langle s_1 \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi \wedge (v \neq 0)) \Downarrow \exists X'.(o = \langle \langle \cdot \rangle_k \langle \rho' \rangle_{env} \rangle \wedge \varphi'), \\ \exists X.(o = \langle \langle s_2 \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi \wedge (v = 0)) \Downarrow \exists X'.(o = \langle \langle \cdot \rangle_k \langle \rho' \rangle_{env} \rangle \wedge \varphi') \end{array}}{\exists X.(o = \langle \langle \text{if}(e) s_1 \text{ else } s_2 \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X'.(o = \langle \langle \cdot \rangle_k \langle \rho' \rangle_{env} \rangle \wedge \varphi')}$$

In this case γ is $\langle \langle \text{if}(e) s_1 \text{ else } s_2 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$, which is equal to $\langle \langle e \rightsquigarrow \text{if}(\square) s_1 \text{ else } s_2 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$, so by Proposition 2 there is some integer i s.t. $\langle \langle e \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle \rightarrow^* \langle \langle i \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$ and $\langle \langle \text{if}(i) s_1 \text{ else } s_2 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle \rightarrow^* \gamma'$, and by LEMMA 9 we know that $i = \theta_\tau(v)$. Now there are two cases to analyze, namely $i \neq 0$ and $i = 0$; we only analyze the former because the later is similar. If $i \neq 0$ then there is only one way to rewrite in one step the configuration containing the computation $\text{if}(i) s_1 \text{ else } s_2$, so we can conclude that $\langle \langle s_1 \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle \rightarrow^* \gamma'$. Since $\theta_\tau \models \varphi \wedge (v \neq 0)$, we can apply the induction hypothesis on the second hypothesis in the rule (ML-IF) and conclude $(\gamma', \tau) \models \exists X'.(o = \langle \langle \cdot \rangle_k \langle \rho' \rangle_{env} \rangle \wedge \varphi')$.

(ML-WHILE)

$$\frac{\begin{array}{l} \exists X.(o = \langle \langle e \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X.(o = \langle \langle v \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi), \\ \exists X.(o = \langle \langle s \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi \wedge (v \neq 0)) \Downarrow \exists X.(o = \langle \langle \cdot \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \end{array}}{\exists X.(o = \langle \langle \text{while}(e) s \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X.(o = \langle \langle \cdot \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi \wedge (v = 0))}$$

In this case we have that $\gamma = \langle \langle \text{while}(e) s \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle \rightarrow^* \gamma'$. We prove by well-founded induction on n that the following proposition holds for any $n > 0$: if $\langle \langle \text{while}(e) s \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle \rightarrow^n \gamma'$ for some $\theta_\tau : Var \rightarrow \mathcal{T}$ such that $\theta_\tau \upharpoonright_{Var \setminus X} = \tau \upharpoonright_{Var \setminus X}$ and $\theta_\tau \models \varphi$, then $(\gamma', \tau) \models \exists X.(o = \langle \langle \cdot \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi \wedge (v = 0))$. Fix an $n > 0$ and suppose that $\langle \langle \text{while}(e) s \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle \rightarrow^n \gamma'$ for some $\theta_\tau : Var \rightarrow \mathcal{T}$. Proposition 2 and LEMMA 9 imply that this rewrite sequence must be of the form $\langle \langle \text{while}(e) s \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle \rightarrow^* \langle \langle \text{if}((\theta_\tau(\rho))(v)) s; \text{while}(e) s \text{ else } \cdot \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle \rightarrow^{n'} \gamma'$, for some $n' \leq n$. We distinguish two cases, like in the case of the conditional. (1) If $(\theta_\tau(\rho))(v) \neq 0$ then the last n' rewrite steps above can only be $\langle \langle \text{if}((\theta_\tau(\rho))(v)) s; \text{while}(e) s \text{ else } \cdot \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle \rightarrow \langle \langle s; \text{while}(e) s \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle \rightarrow^{n'-1} \gamma'$. By Proposition 2 again, it follows that there is some ground environment ρ_0 such that $\langle \langle s \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle \rightarrow^* \langle \langle \cdot \rangle_k \langle \rho_0 \rangle_{env} \rangle$ and $\langle \langle \text{while}(e) s \rangle_k \langle \rho_0 \rangle_{env} \rangle \rightarrow^{n''} \gamma'$ for some $n'' \leq n' - 1$. By the outer induction hypothesis, it follows that $(\langle \langle \cdot \rangle_k \langle \rho_0 \rangle_{env} \rangle, \tau) \models \exists X.(o = \langle \langle \cdot \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi)$, that

is, there is some $\theta'_\tau : Var \rightarrow \mathcal{T}$ with $\theta'_\tau \upharpoonright_{Var \setminus X} = \tau \upharpoonright_{Var \setminus X}$ and $\theta'_\tau \models \varphi$ such that $\rho_0 = \theta'_\tau(\rho)$. Since $\rho_0 = \theta'_\tau(\rho)$ and $n'' < n$ we can apply the inner induction hypothesis and conclude that $(\gamma', \tau) \models \exists X.(o = \langle \langle \cdot \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi \wedge (v = 0))$. (2) If $(\theta_\tau(\rho))(v) = 0$ then it can only be that $n' = 1$ and $\gamma' = \langle \langle \cdot \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \rangle$, so taking θ_τ as the witness for the existential quantifier, we conclude that it is also the case that $(\gamma', \tau) \models \exists X.(o = \langle \langle \cdot \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi \wedge (v = 0))$.

Finally, we prove the soundness of the generic rule. We prove its soundness more generally, for any configuration patterns, even though we only use it for statement patterns.

(ML-CONSEQUENCE)

$$\frac{\models \Gamma \Rightarrow \Gamma_1, \Gamma_1 \Downarrow \Gamma'_1, \models \Gamma'_1 \Rightarrow \Gamma'}{\Gamma \Downarrow \Gamma'}$$

Let $(\gamma, \tau) \models \Gamma$ and suppose that $\text{IMP} \models \gamma \rightarrow^* \gamma'$ for some final γ' . Then also $(\gamma, \tau) \models \Gamma_1$ and, by the induction hypothesis, $(\gamma', \tau) \models \Gamma'_1$, which implies that $(\gamma', \tau) \models \Gamma'$. Q.E.D. THEOREM 8

4.3 Equivalence of Matching Logic and Hoare Logic for IMP

We next show that, in the case of the simple IMP language discussed so far, any property provable using Hoare logic is also provable using matching logic and vice-versa, that is, any property provable using matching is also provable using Hoare logic. Moreover, our proof reductions are mechanical in both directions, which means that one can automatically generate a matching logic proof from any Hoare logic proof and vice-versa. As before, we specialize our proofs for IMP; however, in the embedding of Hoare logic into matching logic part we do not use the fact that the configuration contains only an environment and a computation, so this result also works for other languages that admit Hoare logic proof systems.

Before we proceed with the technical constructions, we need to impose a restriction on the structure of the matching logic patterns to be used throughout this section, more precisely on their environments. Note that matching logic patterns allow us to give more informative specifications than Hoare logic. For example, a pattern of the form $\langle \dots \langle x \mapsto x \rangle_{env} \dots \rangle$ specifies configurations whose environments only declare x (and its value is x), while a pattern $\langle \dots \langle \cdot \rangle_{env} \dots \rangle$ specifies configuration with empty environments; that means that while one is able to derive $\langle \langle x \mapsto x \rangle_{env} \rangle x : = x - x \langle \langle x \mapsto 0 \rangle_{env} \rangle$, it is impossible to derive $\langle \langle \cdot \rangle_{env} \rangle x : = x - x \langle \langle x \mapsto 0 \rangle_{env} \rangle$, simply because one will never be able to “evaluate” x in the empty environment. However, note that the obvious Hoare logic equivalent, namely $\{true\} x : = x - x \{x = 0\}$ is unconditionally derivable.

To prove our desired equivalence result, we fix a finite set of program variables $Z \subset PVar$ which is large enough to include all the program variables that appear in the original program that one wants to verify. Moreover, from here on in this section we assume that all the environments that appear in matching logic patterns have the domain precisely Z . Also, we assume that $Z \subseteq Var$ is a set of “semantic clones” of the program variables in Z , that is, $Z = \{z \mid z \in Z\}$, and that the semantic variables in Z are reserved only for this semantic cloning purpose. Also, let ρ_Z be the special environment mapping each program variable $z \in Z$ into its corresponding semantic clone $z \in Z$.

We first define mappings $H2M$ and $M2H$ taking Hoare logic correctness triples to matching logic correctness pairs, and, respectively, matching logic correctness pairs to Hoare logic correctness triples. Then we show in THEOREM 10 that these mappings are logically inverse to each other and that they take derivable sequents in one logic to derivable sequents in the other logic; for example, if a correctness triple $\{\varphi\} s \{\varphi'\}$ is derivable with the Hoare logic proof system in Figure 3 then the correctness pair $H2M(\{\varphi\} s \{\varphi'\})$ is derivable with the matching logic proof system in Figure 4.

H2M. Assume that variables that appear in Hoare logic specifications but not in the original program are semantic variables in *Var*. Let us first define a homonymous map taking formulae φ and statements s to configuration patterns $H2M(\varphi, s)$ as follows:

$$H2M(\varphi, s) \stackrel{\text{def}}{=} \exists Z. (\circ = \langle \langle s \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi)).$$

Hence, $H2M(\varphi, s)$ is a pattern whose computation is s , whose environment ρ_Z maps each program variable z in φ or in s into its semantic clone z (and possibly many other variables that do not appear in s or in φ), and whose side condition $\rho_Z(\varphi)$ renames all the program variables in the original formula into their semantic counterparts. We now define the mapping from Hoare logic correctness triples into matching logic correctness pairs as follows:

$$H2M(\{\varphi\} s \{\varphi'\}) \stackrel{\text{def}}{=} H2M(\varphi, s) \Downarrow H2M(\varphi', \cdot).$$

For example, if $Z = \{x, z\}$ then

$$\begin{aligned} H2M(\{x > 0 \wedge z = u\} \quad z := x + z \quad \{z > u\}) = \\ \exists x, z. (\circ = \langle \langle z := x + z \rangle_k \langle x \mapsto x, z \mapsto z \rangle_{env} \rangle \wedge x > 0 \wedge z = u) \\ \Downarrow \exists x, z. (\circ = \langle \langle \cdot \rangle_k \langle x \mapsto x, z \mapsto z \rangle_{env} \rangle \wedge z > u). \end{aligned}$$

The resulting matching logic correctness pairs are quite intuitive, making use of pattern bound variables as a bridge between the program variables and the semantic constraints on them.

M2H. Given an environment $\rho = (x_1 \mapsto v_1, x_2 \mapsto v_2, \dots, x_n \mapsto v_n)$, let $\bar{\rho}$ be the FOL formula $x_1 = v_1 \wedge x_2 = v_2 \wedge \dots \wedge x_n = v_n$. Then we define the mapping $M2H$ taking matching logic statement correctness pairs into Hoare logic correctness triples as follows:

$$\begin{aligned} M2H(\exists X. (\circ = \langle \langle s \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X'. (\circ = \langle \langle \cdot \rangle_k \langle \rho' \rangle_{env} \rangle \wedge \varphi')) \\ = \{\exists X. (\bar{\rho} \wedge \varphi)\} s \{\exists X'. (\bar{\rho}' \wedge \varphi')\} \end{aligned}$$

For example, if $\Gamma \Downarrow \Gamma'$ is the correctness pair

$$\exists x, z. (\circ = \langle \langle z := x + z \rangle_k \langle x \mapsto x, z \mapsto z \rangle_{env} \rangle \wedge x > 0 \wedge z = u)$$

$$\Downarrow \exists x, z. (\circ = \langle \langle \cdot \rangle_k \langle x \mapsto x, z \mapsto z \rangle_{env} \rangle \wedge z > u)$$

above, then $M2H(\Gamma \Downarrow \Gamma')$ is the correctness triple

$$\begin{aligned} \{\exists x, z. (x = x \wedge z = z \wedge x > 0 \wedge z = u) \\ z := x + z \\ \{\exists x, z. (x = x \wedge z = z \wedge z > u)\}. \end{aligned}$$

We say that two FOL formulae φ_1 and φ_2 are logically equivalent iff $\models \varphi_1 \Leftrightarrow \varphi_2$. Moreover, correctness triples $\{\varphi_1\} s \{\varphi'_1\}$ and $\{\varphi_2\} s \{\varphi'_2\}$ are *logically equivalent* iff $\models \varphi_1 \Leftrightarrow \varphi_2$ and $\models \varphi'_1 \Leftrightarrow \varphi'_2$; similarly, matching logic correctness pairs $\Gamma_1 \Downarrow \Gamma'_1$ and $\Gamma_2 \Downarrow \Gamma'_2$ are *logically equivalent* iff $\models \Gamma_1 \Leftrightarrow \Gamma_2$ and $\models \Gamma'_1 \Leftrightarrow \Gamma'_2$. Thanks to the rules (HL-CONSEQUENCE) and (ML-CONSEQUENCE), respectively, in both Hoare logic and matching logic, logically equivalent sequents are either both or none derivable. Since $\exists x, z. (x = x \wedge z = z \wedge x > 0 \wedge z = u)$ is logically equivalent to $x > 0 \wedge z = u$ and since $\exists z. (z = z \wedge z > u)$ is logically equivalent to $z > u$, we can conclude that the correctness triple $M2H(\Gamma \Downarrow \Gamma')$ above is logically equivalent to $\{x > 0 \wedge z = u\} \quad z := x + z \quad \{z > u\}$.

THEOREM 10. Any Hoare triple $\{\varphi\} s \{\varphi'\}$ is logically equivalent to $M2H(H2M(\{\varphi\} s \{\varphi'\}))$, and any matching logic correctness pair $\Gamma \Downarrow \Gamma'$ is logically equivalent to $H2M(M2H(\Gamma \Downarrow \Gamma'))$. Moreover, for any Hoare logic proof of $\{\varphi\} s \{\varphi'\}$ one can construct a matching logic proof of $H2M(\{\varphi\} s \{\varphi'\})$, and for any matching logic proof of $\Gamma \Downarrow \Gamma'$ one can construct a Hoare logic proof of $M2H(\Gamma \Downarrow \Gamma')$.

Proof. To prove that $M2H(H2M(\{\varphi\} s \{\varphi'\}))$ is logically equivalent to $\{\varphi\} s \{\varphi'\}$, we need to prove that any φ is logically equivalent to $\exists Z. (\bar{\rho}_Z \wedge \rho_Z(\varphi))$, where Z and ρ_Z are those in the definition of $H2M$. This follows by noting that $\bar{\rho}_Z$ is a conjunct of equalities of the form $z = z$, one equality for each program variable $z \in Z$ (and its corresponding semantic clone $z \in Z$), and noting that $\rho_Z(\varphi)$ replaces each program variable in φ with its semantic clone: one can therefore apply all the equalities $z = z$ backwards on $\rho_Z(\varphi)$ and

thus soundly recover φ , then one can first remove the redundant $\bar{\rho}_Z$ and then the redundant quantifier “ $\exists Z$ ”.

To prove that $H2M \circ M2H$ is logically equivalent to the identity on matching logic correctness pairs, all we need to prove is that any matching logic pattern $\exists Y. (\circ = \langle \langle s \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi)$ is logically equivalent to $\exists X. (\circ = \langle \langle s \rangle_k \langle \rho_X \rangle_{env} \rangle \wedge \rho_X(\exists Y. (\bar{\rho} \wedge \varphi)))$, where X is the set of semantic variable clones of those program variables in the domain of ρ . Since ρ_X only maps program variables $x \in X$ into corresponding semantic clones $x \in X$ which are not used for any other purpose (in particular $X \cap Y = \emptyset$ and φ contains no variable in X), we obtain that $\rho_X(\exists Y. (\bar{\rho} \wedge \varphi)) = \exists Y. (\rho_X(\bar{\rho}) \wedge \varphi)$. Moreover, since the variables in Y can at most be used in ρ and φ , we can move the existential quantifiers in front, that is, $\exists X. (\circ = \langle \langle s \rangle_k \langle \rho_X \rangle_{env} \rangle \wedge \rho_X(\exists Y. (\bar{\rho} \wedge \varphi)))$ is logically equivalent to $\exists Y. \exists X. (\circ = \langle \langle s \rangle_k \langle \rho_X \rangle_{env} \rangle \wedge \rho_X(\bar{\rho}) \wedge \varphi)$. Note that ρ_X is a term of the form $x_1 \mapsto x_2, x_2 \mapsto x_2, x_n \mapsto x_n$ where $X = \{x_1, x_2, \dots, x_n\}$ is the set of program variables in the domain of ρ , that if ρ has the form $x_1 \mapsto v_1, x_2 \mapsto v_2, \dots, x_n \mapsto v_n$ then $\rho_X(\bar{\rho})$ is a formula of the form $x_1 = v_1 \wedge x_2 = v_2 \wedge \dots \wedge x_n = v_n$, and that these are the only places where the semantic variables x_1, x_2, \dots, x_n appear. Replacing each term x_i by the equal term v_i in ρ_X we can systematically replace each mapping $x_i \mapsto x_i$ in ρ_X by a mapping $x_i \mapsto v_i$, thus transforming ρ_X back into ρ . Since the existentially quantified variables in X are not used anymore in the formula except for in the subformula $\rho_X(\bar{\rho})$, we obtain that $\exists Y. \exists X. (\circ = \langle \langle s \rangle_k \langle \rho_X \rangle_{env} \rangle \wedge \rho_X(\bar{\rho}) \wedge \varphi)$ is logically equivalent to $\exists Y. (\circ = \langle \langle s \rangle_k \langle \rho \rangle_{env} \rangle \wedge \exists X. (\rho_X(\bar{\rho}) \wedge \varphi))$. Finally, since $\exists X. (\rho_X(\bar{\rho}))$ is $\exists X. (x_1 = v_1 \wedge x_2 = v_2 \wedge \dots \wedge x_n = v_n)$ which is obviously a tautology, we conclude that $\exists Y. (\circ = \langle \langle s \rangle_k \langle \rho \rangle_{env} \rangle \wedge \exists X. (\rho_X(\bar{\rho}) \wedge \varphi))$ is logically equivalent to $\exists Y. (\circ = \langle \langle s \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi)$.

To prove the remaining of THEOREM 10, we first prove the following lemma stating that the “evaluation” of an expression is unconditionally provable in matching logic:

LEMMA 11. If e is an expression, then the expression correctness pair $\exists X. (\circ = \langle \langle e \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X'. (\circ = \langle \langle v \rangle_k \langle \rho' \rangle_{env} \rangle \wedge \varphi')$ is derivable using the expression rules in the matching logic proof system in Figure 4 iff $X' = X, \rho' = \rho, \varphi' = \varphi$, and ρ is defined in all program variables occurring in e and $v = \rho(e)$.

Proof. The proof of the lemma goes by structural induction on e . There are three cases to analyze.

(1) If e is an integer i , then the correctness pair is derivable iff it is derivable as an instance of rule (ML-INT) in Figure 4, iff $v = i = \rho(i), X' = X, \rho' = \rho$, and $\varphi' = \varphi$.

(2) If e is a variable x , then the correctness pair is derivable iff it is derivable as an instance of rule (ML-LOOKUP) in Figure 4, iff $X' = X$ and $\rho = \rho'$ contains the mapping $x \mapsto v$; if that is the case, then obviously $v = \rho(x)$.

(3) If e is an expression of the form $e_1 \text{ op } e_2$ for some expressions e_1 and e_2 , then the correctness pair is derivable iff it is derivable using the rule (ML-OP) in Figure 4, iff $X' = X$ and $\rho' = \rho$ and $\varphi' = \varphi$ and $v = v_1 \text{ op }_{Int} v_2$ and

$$\exists X. (\circ = \langle \langle e_1 \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X. (\circ = \langle \langle v_1 \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi)$$

$$\exists X. (\circ = \langle \langle e_2 \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X. (\circ = \langle \langle v_2 \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi)$$

are derivable, iff $X' = X$ and $\rho' = \rho$ and $\varphi' = \varphi$ and $v = v_1 \text{ op }_{Int} v_2$ and ρ is defined in all program variables appearing in e_1 and e_2 and $v_1 = \rho(e_1)$ and $v_2 = \rho(e_2)$, iff $X' = X, \rho' = \rho, \varphi' = \varphi$, and ρ is defined in all program variables occurring in e and $v = \rho(e)$ (since $\rho(e_1 \text{ op } e_2) = \rho(e_1) \text{ op }_{Int} \rho(e_2)$).

Q.E.D. LEMMA 11

We now return to THEOREM 10 and prove that for any Hoare logic proof of $\{\varphi\} s \{\varphi'\}$ one can construct a matching logic proof of $H2M(\{\varphi\} s \{\varphi'\})$. The proof goes by structural induction on the formal proof derived using the Hoare logic proof system in Figure 3. We consider each rule in Figure 3 and show how corresponding

matching logic proofs for the hypotheses can be composed into a matching logic proof for the conclusion.

$$\frac{}{\{\varphi[e/x]\} x := e \{\varphi\}} \quad (\text{HL-ASGN})$$

We have to produce a matching logic proof for

$$\exists Z.(\circ = \langle \langle x := e \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi[e/x]))$$

$$\Downarrow \exists Z.(\circ = \langle \langle \cdot \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi)),$$

where Z is the one in the definition of $H2M$. By LEMMA 11, the following is derivable (because Z was chosen large enough to contain all program variables in e , so ρ_Z fits the hypothesis of LEMMA 11)

$$\exists Z.(\circ = \langle \langle e \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi[e/x]))$$

$$\Downarrow \exists Z.(\circ = \langle \langle \rho_Z(e) \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi[e/x])),$$

so by applying rule (ML-ASGN), the following is also derivable:

$$\exists Z.(\circ = \langle \langle x := e \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi[e/x]))$$

$$\Downarrow \exists Z.(\circ = \langle \langle \cdot \rangle_k \langle \rho_Z[\rho_Z(e)/x] \rangle_{env} \rangle \wedge \rho_Z(\varphi[e/x])).$$

Finally, since $\rho_Z(\varphi[e/x]) = \rho_Z[\rho_Z(e)/x](\varphi)$ and since in $\text{FOL}_=$ the formula $\exists Z.(\circ = \langle \langle \cdot \rangle_k \langle \rho_Z[\rho_Z(e)/x] \rangle_{env} \rangle \wedge \rho_Z[\rho_Z(e)/x](\varphi))$ implies the formula $\exists Z.(\circ = \langle \langle \cdot \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi))$ (intuitively, because the former is “more concrete”), we get by rule (ML-CONSEQUENCE) that

$$\exists Z.(\circ = \langle \langle x := e \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi[e/x]))$$

$$\Downarrow \exists Z.(\circ = \langle \langle \cdot \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi))$$

is derivable.

$$\frac{\{\varphi_1\} s_1 \{\varphi_2\}, \{\varphi_2\} s_2 \{\varphi_3\}}{\{\varphi_1\} s_1 ; s_2 \{\varphi_3\}} \quad (\text{HL-SEQ})$$

Suppose $H2M(\varphi_1, s_1) \Downarrow H2M(\varphi_2, \cdot)$ and $H2M(\varphi_2, s_2) \Downarrow H2M(\varphi_3, \cdot)$ are derivable. Desugaring these and the rule (ML-SEQ) in Figure 4, we can easily see that rule (ML-SEQ) implies that the correctness pair $H2M(\varphi_1, s_1 ; s_2) \Downarrow H2M(\varphi_3, \cdot)$ is also derivable.

$$\frac{\{\varphi \wedge (e \neq 0)\} s_1 \{\varphi'\}, \{\varphi \wedge (e = 0)\} s_2 \{\varphi'\}}{\{\varphi\} \text{if } (e) s_1 \text{ else } s_2 \{\varphi'\}} \quad (\text{HL-IF})$$

Suppose, inductively, that the following two correctness pairs

$$\exists Z.(\circ = \langle \langle s_1 \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi \wedge (e \neq 0)))$$

$$\Downarrow \exists Z.(\circ = \langle \langle \cdot \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi'))$$

and

$$\exists Z.(\circ = \langle \langle s_2 \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi \wedge (e = 0)))$$

$$\Downarrow \exists Z.(\circ = \langle \langle \cdot \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi'))$$

are derivable with the proof system in Figure 4. By LEMMA 11

$$\exists Z.(\circ = \langle \langle e \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi))$$

$$\Downarrow \exists Z.(\circ = \langle \langle \rho_Z(e) \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi))$$

is derivable, so (ML-IF) implies that

$$\exists Z.(\circ = \langle \langle \text{if } (e) s_1 \text{ else } s_2 \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi))$$

$$\Downarrow \exists Z.(\circ = \langle \langle \cdot \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi'))$$

is also derivable.

$$\frac{\{\varphi \wedge (e \neq 0)\} s \{\varphi\}}{\{\varphi\} \text{while } (e) s \{\varphi \wedge (e = 0)\}} \quad (\text{HL-WHILE})$$

Suppose, inductively, that the correctness pair

$$\exists Z.(\circ = \langle \langle s \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi \wedge (e \neq 0)))$$

$$\Downarrow \exists Z.(\circ = \langle \langle \cdot \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi))$$

is derivable with the proof system in Figure 4. By LEMMA 11

$$\exists Z.(\circ = \langle \langle e \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi))$$

$$\Downarrow \exists Z.(\circ = \langle \langle \rho_Z(e) \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi))$$

is derivable, so (ML-WHILE) implies that

$$\exists Z.(\circ = \langle \langle \text{while } (e) s \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi))$$

$$\Downarrow \exists Z.(\circ = \langle \langle \cdot \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi \wedge (e = 0)))$$

is also derivable.

$$\frac{\vdash \psi \Rightarrow \varphi, \{\varphi\} s \{\varphi'\}, \vdash \varphi' \Rightarrow \psi'}{\{\psi\} s \{\psi'\}} \quad (\text{HL-CONSEQUENCE})$$

Suppose, inductively, that the correctness pair

$$\exists Z.(\circ = \langle \langle s \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi))$$

$$\Downarrow \exists Z.(\circ = \langle \langle \cdot \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi'))$$

is derivable with the proof system in Figure 4. Since $\vdash \psi \Rightarrow \varphi$ implies $\vdash \rho_Z(\psi) \Rightarrow \rho_Z(\varphi)$, which further implies $\vdash \exists Z.(\circ = \langle \langle s \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\psi)) \Rightarrow \exists Z.(\circ = \langle \langle s \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\varphi))$, and similarly for $\vdash \varphi' \Rightarrow \psi'$, we get by (ML-CONSEQUENCE) that

$$\exists Z.(\circ = \langle \langle s \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\psi))$$

$$\Downarrow \exists Z.(\circ = \langle \langle \cdot \rangle_k \langle \rho_Z \rangle_{env} \rangle \wedge \rho_Z(\psi'))$$

is also derivable with the proof system in Figure 4.

We now prove that for any matching logic proof of $\Gamma \Downarrow \Gamma'$ one can construct a Hoare logic proof of $M2H(\Gamma \Downarrow \Gamma')$. The proof goes by structural induction on the formal proof derived using the matching logic proof system in Figure 4. We consider each rule in Figure 4 in desugared form and show how corresponding Hoare logic proofs for the hypotheses can be composed into a matching logic proof for the conclusion.

(ML-ASGN)

$$\frac{\exists X.(\circ = \langle \langle e \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X.(\circ = \langle \langle v \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi)}{\exists X.(\circ = \langle \langle x := e \rangle_k \langle \rho \rangle_{env} \rangle \wedge \varphi) \Downarrow \exists X.(\circ = \langle \langle \cdot \rangle_k \langle \rho[v/x] \rangle_{env} \rangle \wedge \varphi)}$$

Since we can assume that have a matching logic proof for the hypothesis of (ML-ASGN), then $v = \rho(e)$ by LEMMA 11, so $M2H(\Gamma \Downarrow \Gamma')$ is the Hoare triple $\{\exists X.(\overline{\rho} \wedge \varphi)\} x := e \{\exists X.(\overline{\rho}[\rho(e)/x] \wedge \varphi)\}$. We show this derivable using the Hoare logic proof system in Figure 3. Note that $\{(\exists X.(\overline{\rho}[\rho(e)/x] \wedge \varphi))[e/x]\} x := e \{\exists X.(\overline{\rho}[\rho(e)/x] \wedge \varphi)\}$, is derivable by rule (HL-ASGN). We next show that $\vdash \exists X.(\overline{\rho} \wedge \varphi) \Rightarrow (\exists X.(\overline{\rho}[\rho(e)/x] \wedge \varphi))[e/x]$, so that the desired result holds by an application of the rule (HL-CONSEQUENCE). First, note that φ only uses semantic variables, so $\varphi[e/x] = \varphi$. Second, let $\rho \setminus x$ be the environment ρ from which the mapping “ $x \mapsto _$ ” is removed; thus, $\overline{\rho}[\rho(e)/x] = \overline{\rho \setminus x} \wedge \rho(e) = e$. Since there is no name capturing between the semantic variables in X and the program variables in the substitution $[e/x]$, we get that $(\exists X.(\overline{\rho}[\rho(e)/x] \wedge \varphi))[e/x] = \exists X.(\overline{\rho \setminus x} \wedge \rho(e) = e \wedge \varphi)$, so it suffices to show that $\vdash \exists X.(\overline{\rho} \wedge \varphi) \Rightarrow \exists X.(\overline{\rho \setminus x} \wedge \rho(e) = e \wedge \varphi)$, and in particular that $\vdash \overline{\rho} \Rightarrow \overline{\rho \setminus x} \wedge \rho(e) = e$, and since obviously $\vdash \overline{\rho} \Rightarrow \overline{\rho \setminus x}$, all we have to show is that $\vdash \overline{\rho} \Rightarrow \rho(e) = e$. Fortunately, this follows easily by equational reasoning, replacing each program variable in e with its semantic counterpart in $\overline{\rho}$ until eventually e becomes $\rho(e)$.

(ML-SEQ)

$$\frac{\begin{array}{l} \exists X_1.(\circ = \langle \langle s_1 \rangle_k \langle \rho_1 \rangle_{env} \rangle \wedge \varphi_1) \Downarrow \exists X_2.(\circ = \langle \langle \cdot \rangle_k \langle \rho_2 \rangle_{env} \rangle \wedge \varphi_2), \\ \exists X_2.(\circ = \langle \langle s_2 \rangle_k \langle \rho_2 \rangle_{env} \rangle \wedge \varphi_2) \Downarrow \exists X_3.(\circ = \langle \langle \cdot \rangle_k \langle \rho_3 \rangle_{env} \rangle \wedge \varphi_3) \end{array}}{\exists X_1.(\circ = \langle \langle s_1 ; s_2 \rangle_k \langle \rho_1 \rangle_{env} \rangle \wedge \varphi_1) \Downarrow \exists X_3.(\circ = \langle \langle \cdot \rangle_k \langle \rho_3 \rangle_{env} \rangle \wedge \varphi_3)}$$

Suppose, inductively, that we already have Hoare logic proofs for the triple $\{\exists X_1.(\overline{\rho_1} \wedge \varphi_1)\} s_1 \{\exists X_2.(\overline{\rho_2} \wedge \varphi_2)\}$ and for the triple $\{\exists X_2.(\overline{\rho_2} \wedge \varphi_2)\} s_2 \{\exists X_3.(\overline{\rho_3} \wedge \varphi_3)\}$ corresponding to the two hypotheses in rule (ML-SEQ). Then by rule (HL-SEQ) we can also build a Hoare logic proof for $\{\exists X_1.(\overline{\rho_1} \wedge \varphi_1)\} s_1 ; s_2 \{\exists X_3.(\overline{\rho_3} \wedge \varphi_3)\}$.

(ML-IF)

$$\begin{aligned} & \exists X.(\circ = \langle\langle e \rangle_k \langle \rho \rangle_{env}\rangle \wedge \varphi) \Downarrow \exists X.(\circ = \langle\langle v \rangle_k \langle \rho \rangle_{env}\rangle \wedge \varphi), \\ & \exists X.(\circ = \langle\langle s_1 \rangle_k \langle \rho \rangle_{env}\rangle \wedge \varphi \wedge (v \neq 0)) \Downarrow \exists X'.(\circ = \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle \wedge \varphi'), \\ & \exists X.(\circ = \langle\langle s_2 \rangle_k \langle \rho \rangle_{env}\rangle \wedge \varphi \wedge (v = 0)) \Downarrow \exists X'.(\circ = \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle \wedge \varphi') \\ & \exists X.(\circ = \langle\langle \text{if}(e) s_1 \text{ else } s_2 \rangle_k \langle \rho \rangle_{env}\rangle \wedge \varphi) \Downarrow \exists X'.(\circ = \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle \wedge \varphi') \end{aligned}$$

By LEMMA 11, $v = \rho(e)$. Suppose, inductively, that the triple $\{\exists X.(\bar{\rho} \wedge \varphi \wedge (\rho(e) \neq 0))\} s_1 \{ \exists X'.(\bar{\rho}' \wedge \varphi') \}$ as well as the triple $\{\exists X.(\bar{\rho} \wedge \varphi \wedge (\rho(e) = 0))\} s_2 \{ \exists X'.(\bar{\rho}' \wedge \varphi') \}$ are derivable using the Hoare logic proof system in Figure 3. We can use the equalities in $\bar{\rho}$ to systematically replace each program variable in e with its corresponding semantic value in ρ until e becomes $\rho(e)$, so one can show that $\models \bar{\rho} \wedge \varphi \wedge e \neq 0 \Rightarrow \bar{\rho} \wedge \varphi \wedge \rho(e) \neq 0$ and $\models \bar{\rho} \wedge \varphi \wedge e = 0 \Rightarrow \bar{\rho} \wedge \varphi \wedge \rho(e) = 0$. Extending these to the quantified expressions and using the rule (HL-CONSEQUENCE) twice, we get that the triples $\{\exists X.(\bar{\rho} \wedge \varphi \wedge e \neq 0)\} s_1 \{ \exists X'.(\bar{\rho}' \wedge \varphi') \}$ and $\{\exists X.(\bar{\rho} \wedge \varphi \wedge e = 0)\} s_2 \{ \exists X'.(\bar{\rho}' \wedge \varphi') \}$ are derivable using the Hoare logic proof system in Figure 3, so by (HL-IF) the triple $\{\exists X.(\bar{\rho} \wedge \varphi)\} \text{if}(e) s_1 \text{ else } s_2 \{ \exists X'.(\bar{\rho}' \wedge \varphi') \}$ is also derivable.

(ML-WHILE)

$$\begin{aligned} & \exists X.(\circ = \langle\langle e \rangle_k \langle \rho \rangle_{env}\rangle \wedge \varphi) \Downarrow \exists X.(\circ = \langle\langle v \rangle_k \langle \rho \rangle_{env}\rangle \wedge \varphi), \\ & \exists X.(\circ = \langle\langle s \rangle_k \langle \rho \rangle_{env}\rangle \wedge \varphi \wedge (v \neq 0)) \Downarrow \exists X.(\circ = \langle\langle \cdot \rangle_k \langle \rho \rangle_{env}\rangle \wedge \varphi) \\ & \exists X.(\circ = \langle\langle \text{while}(e) s \rangle_k \langle \rho \rangle_{env}\rangle \wedge \varphi) \Downarrow \exists X.(\circ = \langle\langle \cdot \rangle_k \langle \rho \rangle_{env}\rangle \wedge \varphi \wedge (v = 0)) \end{aligned}$$

By LEMMA 11, $v = \rho(e)$. Suppose, inductively, that the triple $\{\exists X.(\bar{\rho} \wedge \varphi \wedge (\rho(e) \neq 0))\} s \{ \exists X.(\bar{\rho} \wedge \varphi) \}$ is derivable using the Hoare logic proof system in Figure 3. Like above, we can use the equalities in $\bar{\rho}$ to show that $\models \bar{\rho} \wedge \varphi \wedge e \neq 0 \Rightarrow \bar{\rho} \wedge \varphi \wedge \rho(e) \neq 0$, so by rule (HL-CONSEQUENCE) noting that the variables in e and X are disjoint, we get that $\{\exists X.(\bar{\rho} \wedge \varphi) \wedge e \neq 0\} s \{ \exists X.(\bar{\rho} \wedge \varphi) \}$ is derivable using the Hoare logic proof system in Figure 3, so by (HL-WHILE) the triple $\{\exists X.(\bar{\rho} \wedge \varphi)\} \text{while}(e) s \{ \exists X.(\bar{\rho} \wedge \varphi) \wedge e = 0 \}$ is also derivable. Finally, like above we can show that $\models \exists X.(\bar{\rho} \wedge \varphi) \wedge e = 0 \Rightarrow \exists X.(\bar{\rho} \wedge \varphi \wedge \rho(e) = 0)$, so the result follows by one more application of the rule (HL-CONSEQUENCE).

(ML-CONSEQUENCE)

$$\begin{aligned} & \models \exists X.(\circ = \langle\langle s \rangle_k \langle \rho \rangle_{env}\rangle \wedge \varphi) \Rightarrow \exists X_1.(\circ = \langle\langle s \rangle_k \langle \rho_1 \rangle_{env}\rangle \wedge \varphi_1), \\ & \exists X_1.(\circ = \langle\langle s \rangle_k \langle \rho_1 \rangle_{env}\rangle \wedge \varphi_1) \Downarrow \exists X'_1.(\circ = \langle\langle \cdot \rangle_k \langle \rho'_1 \rangle_{env}\rangle \wedge \varphi'_1), \\ & \models \exists X'_1.(\circ = \langle\langle \cdot \rangle_k \langle \rho'_1 \rangle_{env}\rangle \wedge \varphi'_1) \Rightarrow \exists X'.(\circ = \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle \wedge \varphi') \\ & \exists X.(\circ = \langle\langle s \rangle_k \langle \rho \rangle_{env}\rangle \wedge \varphi) \Downarrow \exists X'.(\circ = \langle\langle \cdot \rangle_k \langle \rho' \rangle_{env}\rangle \wedge \varphi') \end{aligned}$$

Suppose, inductively, that $\{\exists X_1.(\bar{\rho}_1 \wedge \varphi_1)\} s \{ \exists X'_1.(\bar{\rho}'_1 \wedge \varphi'_1) \}$ is derivable using the Hoare logic proof system in Figure 3. We show that $\models \exists X.(\bar{\rho} \wedge \varphi) \Rightarrow \exists X_1.(\bar{\rho}_1 \wedge \varphi_1)$ and $\models \exists X'_1.(\bar{\rho}'_1 \wedge \varphi'_1) \Rightarrow \exists X'.(\bar{\rho}' \wedge \varphi')$, so that the result follows by applying rule (HL-CONSEQUENCE). The two properties are similar, so we only prove the first one. Let $\xi : \text{Var} \cup \text{PVar} \rightarrow \mathcal{T}$ be a valuation such that $\xi \models \exists X.(\bar{\rho} \wedge \varphi)$. All we need to show is that $\xi \models \exists X_1.(\bar{\rho}_1 \wedge \varphi_1)$. Let $\theta_\xi : \text{Var} \cup \text{PVar} \rightarrow \mathcal{T}$ be a valuation such that $\theta_\xi \upharpoonright_{(\text{Var} \cup \text{PVar}) \setminus X} = \xi \upharpoonright_{(\text{Var} \cup \text{PVar}) \setminus X}$ and $\theta_\xi \models \bar{\rho} \wedge \varphi$. Let $\tau : \text{Var} \rightarrow \mathcal{T}$ be $\xi \upharpoonright_{\text{Var}}$ and let $\theta_\tau : \text{Var} \rightarrow \mathcal{T}$ be $\theta_\xi \upharpoonright_{\text{Var}}$. Note that $\theta_\tau \upharpoonright_{\text{Var} \setminus X} = \tau \upharpoonright_{\text{Var} \setminus X}$ and $\theta_\tau \models \varphi$. Thus, if we take γ to be the configuration $\langle\langle s \rangle_k \langle \theta_\tau(\rho) \rangle_{env}\rangle$, then θ_τ is a witness that $(\gamma, \tau) \models \exists X.(\circ = \langle\langle s \rangle_k \langle \rho \rangle_{env}\rangle \wedge \varphi)$. By hypothesis we get that $(\gamma, \tau) \models \exists X_1.(\circ = \langle\langle s \rangle_k \langle \rho_1 \rangle_{env}\rangle \wedge \varphi_1)$, so there is some $\theta^1_\tau : \text{Var} \rightarrow \mathcal{T}$ such that $\theta^1_\tau \upharpoonright_{\text{Var} \setminus X_1} = \tau \upharpoonright_{\text{Var} \setminus X_1}$, $\theta^1_\tau \models \varphi_1$, and $\gamma = \langle\langle s \rangle_k \langle \theta^1_\tau(\rho_1) \rangle_{env}\rangle$, the latter being equivalent to $\theta^1_\tau(\rho_1) = \theta_\tau(\rho)$. Let $\theta^1_\xi : \text{Var} \cup \text{PVar} \rightarrow \mathcal{T}$ be the valuation with $\theta^1_\xi \upharpoonright_{\text{Var}} = \theta^1_\tau$ and $\theta^1_\xi \upharpoonright_{\text{PVar}} = \theta_\xi \upharpoonright_{\text{PVar}} = \xi \upharpoonright_{\text{PVar}}$. Then note that $\theta^1_\xi \models \bar{\rho}_1 \wedge \varphi_1$, so $\xi \models \exists X_1.(\bar{\rho}_1 \wedge \varphi_1)$. Q.E.D. THEOREM 10

Start with the executable \mathbb{K} semantics of IMP in (Figure 2) and:

(1) Define patterns (in sugared form), by adding:	(language-independent)
syntactic categories <i>Var</i> and <i>Form</i> for semantic variables and formulae, and $\text{CfgItem} ::= \dots \mid \langle \text{Set} \vdash [\text{Var}] \rangle_{bnd} \mid \langle \text{Form} \rangle_{form}$	
(2) Extend language syntax with pattern annotations and add a new top cell $S ::= \dots \mid \text{assert } \text{Cfg}$ and $\text{Top} ::= \langle \text{Set} \vdash [\text{Cfg}] \rangle_\tau$	
(3) Add language-independent, verification-framework-specific rules:	
$\langle\langle \text{assert } \Gamma \sim k \rangle_k c \rangle \rightarrow \Gamma_{(k)_k}$ when $\langle\langle k \rangle_k c \rangle \Rightarrow \Gamma_{(k)_k}$	(V-ASSERTION-CHECKING)
$\Gamma \rightarrow \Gamma_1 \Gamma_2$ when $\models \Gamma \Rightarrow \Gamma_1 \vee \Gamma_2$	(V-CASE-AND-ABSTRACTION)
$\langle\langle \cdot \rangle_k c \rangle \rightarrow \cdot$	(V-done)
$\langle\langle \text{false} \rangle_{form} c \rangle \rightarrow \cdot$	(V-infeasible)
(4) Replace rules for <i>if</i> and <i>while</i> with:	(language-specific)– here IMP
$\langle\langle \text{if}(v) s_1 \text{ else } s_2 \rangle_k \langle \varphi \rangle_{form} c \rangle$	(V-IF)
$\rightarrow \langle\langle s_1 \sim k \rangle_k \langle \varphi \wedge v \neq 0 \rangle_{form} c \rangle \quad \langle\langle s_2 \sim k \rangle_k \langle \varphi \wedge v = 0 \rangle_{form} c \rangle$	
$\langle\langle \text{while}(e) s \rangle_k \langle \rho \rangle_{env} c \rangle \rightarrow \langle\langle \text{if}(e) (s; \text{assert}(c)) \text{ else } k \rangle_k c \rangle$	(V-WHILE)

Figure 6. Matching logic verifier, generic(1,2,3) and IMP-specific(4)

5. Deriving an IMP Matching Logic Verifier in \mathbb{K}

Figure 6 shows how we can turn the \mathbb{K} semantics of IMP into a program verifier in four simple steps, the first three being language-independent (so reusable across different languages). Section 6 shows that it is equally simple to do the same for more complex languages with a heap, dynamic memory allocation/deallocation, and even arbitrary pointer arithmetic. We next discuss each of the four steps deriving the prover in Figure 6 and show that it is *sound* and *complete* for matching logic (provided enough annotations). Intuitively, the idea of our approach is to execute the \mathbb{K} semantics symbolically on a pattern annotated variant of the original program, splitting the verification task in two tasks at each conditional. This way, one can have in principle a path explosion problem, but, as we show, that can be mitigated by providing additional annotations.

The first step is to add algebraic infrastructure for patterns. It is easier and more intuitive to work with patterns in sugared notation as bags, like in NOTATION 6. All we need to do is to add two new subcells, one holding the set of bound variables and the other holding the constraints. However, one needs to also add all the syntax of $\text{FOL}_=$ formulae, but we assume that to be defined once and for all and import it in each language definition. We therefore assume we are given the syntactic categories *Var* for logical variables and *Form* for $\text{FOL}_=$ formulae. Since our $\text{FOL}_=$ is many-sorted, *Var* is also many-sorted: Var_{Int} , Var_{Env} , etc. Moreover, since logical variables act as symbolic values, we need to subsort them to the domains of values: $\text{Var}_{Int} < \text{Int}$, $\text{Var}_{Env} < \text{Env}$, etc. Note that terms of sort *Cfg* of the form $\langle\langle k \rangle_k \langle \rho \rangle_{env} \langle \varphi \rangle_{form} \langle X \rangle_{bnd} \rangle$ are now patterns, while terms of sort *Cfg* of the form $\langle\langle k \rangle_k \langle \rho \rangle_{env} \rangle$ with ρ ground are still configurations; in this section we only work with patterns, though. Thanks to \mathbb{K} 's modularity, equations and rules from Figure 2 apply *unchanged* on patterns as well, yielding a symbolic execution engine for IMP for free. However, this simplistic symbolic engine may get stuck at control statements. We address this problem below by splitting the symbolic execution in two executions, one for each branch, accumulating the corresponding constraints.

The second step is to add program annotations. We do this by allowing the user to state a partial pattern at any place in the computation; by “partial” pattern we mean one whose $\langle \dots \rangle_k$ cell is omitted (it is unnecessary, since the current computation is understood). We also add a new top-level cell wrapping a set of patterns, each corresponding to one symbolic execution path that is being explored. The configuration of the program verifier will therefore consist of

a cell $\langle \Gamma_1 \Gamma_2 \dots \Gamma_n \rangle_\top$, each Γ_i being a pattern with annotated computation. Interestingly, the equations and rules in Figure 2 work with such extended configurations without any change, because they can operate “in parallel” in each of the embedded patterns.

The third step is to introduce some generic rules that are the same for all languages. (V-ASSERTION-CHECKING) gives semantics to pattern assertions: if pattern assertion Γ is reached, then it should hold in the current specification. The notation $\Gamma_{\langle k \rangle_k}$ stands for the complete pattern obtained by infusing cell $\langle k \rangle_k$ into the partial pattern Γ , that is, $\langle c \rangle_{\langle k \rangle_k} =_{\text{def}} \langle \langle k \rangle_k c \rangle$. Note that we switch to the asserted pattern Γ for the rest of the verification task and discard c , the rationale being that one’s intention when asserting a pattern at a place in a program is to provide details for the remaining proof tasks; this is particularly important when Γ is a loop invariant, i.e., an assertion preceding a loop. In practice, providing *all* the details for the remaining proof tasks may be inconvenient, so one may want instead to keep c combined with useful information from Γ . Rule (V-CASE-AND-ABSTRACTION) allows one to rewrite a verification task to a set of potentially simpler tasks. This rule is more general than theoretically needed for the soundness and completeness of our verifier w.r.t. the matching logic proof system (theoretically, all that is needed is an abstraction rule “ $\Gamma \rightarrow \Gamma'$ when $\models \Gamma \Rightarrow \Gamma'$ ”², obtained when $\Gamma_1 = \Gamma_2 = \Gamma'$); however, we found it very useful in practice to split cases. To see why this rule is correct, note that it corresponds to the following (sound) matching logic proof rule:

$$\frac{\models \Gamma \Rightarrow \Gamma_1 \vee \Gamma_2, \quad \Gamma_1 \Downarrow \Gamma', \quad \Gamma_2 \Downarrow \Gamma'}{\Gamma \Downarrow \Gamma'} \quad (\text{ML-CASE}) \quad (\text{only statement patterns})$$

In implementations, one should use (V-CASE-AND-ABSTRACTION) conservatively, because it may lead to nontermination (when $\Gamma_1 = \Gamma_2$ and $\Gamma \Leftrightarrow \Gamma_1$). We also add rules (V-DONE) and (V-INFEASIBLE) discharging verified tasks; a task is considered verified either when its computation is empty, meaning that all the assertions stated as annotations have been proved, or when it is found infeasible, meaning that its constraints yield *false*. These last two rules eventually empty the top level cell when the program is fully verified.

All three steps above are language-independent: they can be reused for any language, so they can be regarded as part of the matching logic verification framework. The fourth (and last) step towards developing a matching logic program verifier for a language is to modify its \mathbb{K} semantics for control constructs to deal with symbolic configurations. The strictness equations of the conditional will ensure that its condition is evaluated to a symbolic value v (i.e., one containing only domain operations and semantic variables in *Var*). However, since the current pattern constraints φ may not have enough information about the validity of v , we split the current task into two tasks, one for each branch, adding to φ the corresponding assumption about v . This is the only rule that increases the number of verification tasks in the top cell. The rule for while is tricky but clear once one assumes that the current pattern acts as invariant (such an invariant may be given via an ordinary pattern assertion right before the loop, which will be checked and then assumed by first rule in step three above): the resulting conditional generates two branches, one in which the loop condition is assumed true and so the loop body is processed followed by an assertion stating the invariant holds, and one assuming the loop condition false and the remaining computation k is processed.

The above completes the discussion on the derivation of our IMP prover in Figure 6 from the \mathbb{K} semantics of IMP in Figure 2. To use the prover, one needs to provide a program annotated with pattern assertions and an initial pattern. The rewrite system in

² The intuition for the abstraction rule would be that one can at any moment drop some of the information about the current program configuration (e.g., irrelevant axioms, etc.); if one can still check all the remaining assertions with less information, then one can also check them with more information.

Figure 6 is then launched on the top cell containing initially only one pattern, namely the initial pattern infused with the program. The program is considered fully verified when the top cell becomes empty. Let us illustrate our verifier on the sum program with its pre- and post-condition patterns, as given in Figure 5. This program is simple enough that we only need to assert a loop invariant. Therefore, let us make the following notations:

$$\begin{aligned} \Gamma_{\text{start}} &\equiv \langle \langle s=0; n=1; \text{assert } \langle c_{\text{inv}} \rangle; \\ &\quad \text{while } (n \neq p+1) \{ s=s+n; n=n+1 \}; \text{assert } \langle c_{\text{post}} \rangle \rangle_k c_{\text{pre}} \rangle \\ c_{\text{pre}} &\equiv \langle p \mapsto p, s \mapsto s, n \mapsto n, \rho \rangle_{\text{env}} c \langle p \geq 0 \rangle_{\text{form}} \langle \cdot \rangle_{\text{bnd}} \\ c_{\text{post}} &\equiv \langle p \mapsto p, s \mapsto p(p+1)/2, n \mapsto p+1, \rho \rangle_{\text{env}} c \langle p \geq 0 \rangle_{\text{form}} \langle \cdot \rangle_{\text{bnd}} \\ c_{\text{inv}} &\equiv \langle p \mapsto p, s \mapsto n(n-1)/2, n \mapsto n, \rho \rangle_{\text{env}} c \langle p \geq 0 \wedge n \leq p+1 \rangle_{\text{form}} \langle n \rangle_{\text{bnd}} \end{aligned}$$

Letting the rewrite engine run on the top configuration of the verifier containing the pattern $\langle \Gamma_{\text{start}} \rangle_\top$, it produces the rewrite sequence $\langle \Gamma_{\text{start}} \rangle_\top \rightarrow^* \langle \Gamma_1 \rangle_\top \rightarrow \langle \Gamma_2 \rangle_\top \rightarrow \langle \Gamma_3 \rangle_\top \rightarrow \langle \Gamma_4 \Gamma_6 \rangle_\top \rightarrow^* \langle \Gamma_5 \Gamma_6 \rangle_\top \rightarrow^* \langle \cdot \rangle_\top$, where

$$\begin{aligned} \Gamma_1 &\equiv \langle \langle \text{assert } \langle c_{\text{inv}} \rangle; \text{while } (n \neq p+1) \{ s=s+n; n=n+1 \}; \text{assert } \langle c_{\text{post}} \rangle \rangle_k c_1 \rangle \\ c_1 &\equiv \langle p \mapsto p, s \mapsto 0, n \mapsto 1, \rho \rangle_{\text{env}} c \langle p \geq 0 \rangle_{\text{form}} \langle \cdot \rangle_{\text{bnd}} \\ \Gamma_2 &\equiv \langle \langle \text{while } (n \neq p+1) \{ s=s+n; n=n+1 \}; \text{assert } \langle c_{\text{post}} \rangle \rangle_k c_{\text{inv}} \rangle \\ \Gamma_3 &\equiv \langle \langle \text{if } (n \neq p+1) \{ s=s+n; n=n+1; \text{assert } \langle c_{\text{inv}} \rangle \} \text{ else } \text{assert } \langle c_{\text{post}} \rangle \} \rangle_k c_{\text{inv}} \rangle \\ \Gamma_4 &\equiv \langle \langle s=s+n; n=n+1; \text{assert } \langle c_{\text{inv}} \rangle \rangle_k c_4 \rangle \\ c_4 &\equiv \langle p \mapsto p, s \mapsto n(n-1)/2, n \mapsto n, \rho \rangle_{\text{env}} c \langle p \geq 0 \wedge n < p+1 \rangle_{\text{form}} \langle n \rangle_{\text{bnd}} \\ \Gamma_5 &\equiv \langle \langle \langle c_{\text{inv}} \rangle \rangle_k c_5 \rangle \\ c_5 &\equiv \langle p \mapsto p, s \mapsto n(n+1)/2, n \mapsto n+1, \rho \rangle_{\text{env}} c \langle p \geq 0 \wedge n < p+1 \rangle_{\text{form}} \langle n \rangle_{\text{bnd}} \\ \Gamma_6 &\equiv \langle \langle \langle c_{\text{post}} \rangle \rangle_k c_6 \rangle \\ c_6 &\equiv \langle p \mapsto p, s \mapsto n(n-1)/2, n \mapsto n, \rho \rangle_{\text{env}} c \langle p \geq 0 \wedge n = p+1 \rangle_{\text{form}} \langle n \rangle_{\text{bnd}} \end{aligned}$$

Note that the top-level set of patterns eventually emptied, meaning that all the pattern assertions have been checked on all the paths. There are three assertion checks encountered in the run of the verifier above (corresponding to the side condition of the assertion checking rule in step three in Figure 6), two for the invariant and one for the postcondition. All these are easy to check both manually and automatically (using SMT solvers, as our prover does).

THEOREM 12. (Soundness and completeness of \mathbb{K} verifier w.r.t. matching logic) *The following hold, where given an annotated computation $s \in S$ like in Figure 6, $\bar{s} \in S$ is the computation obtained by removing all pattern assertions from s :*

1. **(Soundness)** *If $\langle \langle \langle s; \text{assert } \langle c_{\text{post}} \rangle \rangle_k c_{\text{pre}} \rangle \rangle_\top \rightarrow^* \langle \cdot \rangle_\top$ using the \mathbb{K} definition in Figure 6, then $\langle c_{\text{pre}} \rangle \bar{s} \langle c_{\text{post}} \rangle$ is derivable using the matching logic proof system in Figure 4;*
2. **(Completeness)** *If $\langle c_{\text{pre}} \rangle s \langle c_{\text{post}} \rangle$ is derivable using the matching logic proof system in Figure 4, then there is some annotated computation \bar{s} such that $\bar{s} = s$ and $\langle \langle \langle s; \text{assert } \langle c_{\text{post}} \rangle \rangle_k c_{\text{pre}} \rangle \rangle_\top \rightarrow^* \langle \cdot \rangle_\top$ using the \mathbb{K} definition in Figure 6.*

Proof. (Soundness) Note that the \mathbb{K} definition in Figure 6 takes the original term $\langle \langle \langle s; \text{assert } \langle c_{\text{post}} \rangle \rangle_k c_{\text{pre}} \rangle \rangle_\top \rightarrow^* \langle \cdot \rangle_\top$ and iteratively rewrites it to a sequence of terms which are AC soups (terms modulo associativity and commutativity) of the form $\langle \Gamma_1, \Gamma_2, \dots, \Gamma_n \rangle_\top$. Moreover, there are no rewrite rules that match two or more such Γ ’s in a soup, which means that we can serialize the rewrite steps so that at each moment precisely one Γ in the soup is being rewritten (recall that rewrite logic allows parallel rewriting). Note that there are rules that rewrite a pattern Γ to another pattern, others that rewrite a pattern to two patterns, and others that dissolve a pattern (rewrite it to the empty set). Moreover, it is almost always the case that for a pattern Γ in the soup, there is only a precisely determined sequence of rewrite steps that can be applied on it; since the set of patterns is eventually emptied we can assume that such uniquely determined sequences of rewrite steps will take place. For example, if a pattern Γ in the soup is satisfiable (i.e., it is not logically equivalent to *false*) and has the form $\langle \langle e \curvearrowright k \rangle_k \langle \rho \rangle_{\text{env}} c \rangle$, then this pattern will eventually become $\langle \langle \rho(e) \curvearrowright k \rangle_k \langle \rho \rangle_{\text{env}} c \rangle$. We let the (tedious

but easy) precise proof of this intuitive fact as an exercise to the interested reader (hint: the proof is similar to that of PROPOSITION 2). Suppose $\Gamma \rightsquigarrow \Gamma_s$, where Γ_s is a set of patterns, is such a possible “determined rewriting” relation. Then the discussion above implies that $(\Gamma)_{\top} \rightarrow^* \langle \cdot \rangle_{\top}$ if and only if $(\Gamma)_{\top} \rightsquigarrow^* \langle \cdot \rangle_{\top}$. Our proof strategy now is as follows:

1. Pick an appropriate “determined rewrite” relation \rightsquigarrow as above;
2. Show that all the patterns Γ' that will ever be obtained by applications of the new rewrite relation \rightsquigarrow starting with the original pattern $\langle \langle c_{pre} \langle s \rightsquigarrow \text{assert}(c_{post}) \rangle_k \rangle_{\top} \rangle_{\top}$ have a similar form, i.e., $\Gamma' = \langle \langle c'_{pre} \langle s' \rightsquigarrow \text{assert}(c'_{post}) \rangle_k \rangle_{\top} \rangle_{\top}$ for some c'_{pre} , s' , c'_{post} , where s' is either a statement or an empty computation;
3. To each such pattern Γ' of the form above, we associate a correctness pair $\langle c'_{pre} \rangle_{\top} s' \langle c'_{post} \rangle_{\top}$ when s' is a statement and the formula $\langle \langle \cdot \rangle_k c'_{pre} \rangle \Rightarrow \langle \langle \cdot \rangle_k c'_{post} \rangle$ when s' is empty, written $\overline{\Gamma'}$;
4. For each $\Gamma \rightsquigarrow \Gamma_s$, we prove that if $\overline{\Gamma'}$ is derivable using the matching logic proof system in Figure 4 for each $\Gamma' \in \Gamma_s$, then $\overline{\Gamma}$ is also derivable; by abuse of terminology, a formula is called derivable iff it is a tautology.

It is easy to see that the four steps above imply the desired soundness result. Let us do all four steps above in parallel, by inductively defining such a relation \rightsquigarrow and showing that it has the desired properties. The original pattern $\langle \langle c_{pre} \langle s; \text{assert}(c_{post}) \rangle_k \rangle_{\top} \rangle_{\top}$ plays no role from here on in the soundness proof, so we reuse its symbols. Let us first define \rightsquigarrow on unsatisfiable patterns:

- $\langle \langle \text{false} \rangle_{\text{form}} \langle s \rightsquigarrow \text{assert}(c_{post}) \rangle_k c \rangle_{\top} \rightsquigarrow \cdot$

Note that on such unsatisfiable patterns Γ , it is indeed the case that “ $\Gamma \rightarrow^* \cdot$ ” if and only if “ $\Gamma \rightsquigarrow^* \cdot$ ”. Also, note that the correctness pairs associated to unsatisfiable patterns like in 3 above are indeed derivable with the matching logic proof system, since they have the form $\text{false} \Downarrow \Gamma'$ and these obviously sound correctness pairs are derivable (we let it as exercise).

Let us now define \rightsquigarrow on case splits exactly like \rightarrow :

- $\Gamma \rightsquigarrow \Gamma_1 \Gamma_2$ iff $\Gamma \rightarrow \Gamma_1 \Gamma_2$

Since $\models \Gamma \Rightarrow \Gamma_1 \vee \Gamma_2$, all three patterns must have the same computation. Suppose that $\overline{\Gamma_1}$ and $\overline{\Gamma_2}$ are derivable, and that

- Γ is $\langle \langle s \rightsquigarrow \text{assert}(c') \rangle_k c \rangle_{\top}$,
- Γ_1 is $\langle \langle s \rightsquigarrow \text{assert}(c') \rangle_k c_1 \rangle_{\top}$, and
- Γ_2 is $\langle \langle s \rightsquigarrow \text{assert}(c') \rangle_k c_2 \rangle_{\top}$.

Therefore, $\langle c_1 \rangle_{\top} s \langle c' \rangle_{\top}$ and $\langle c_2 \rangle_{\top} s \langle c' \rangle_{\top}$ are derivable. Then by rule (ML-CASES) it follows that $\langle c \rangle_{\top} s \langle c' \rangle_{\top}$ is also derivable.

Let us now assume that $\Gamma = \langle \langle c_{pre} \langle s \rightsquigarrow \text{assert}(c_{post}) \rangle_k \rangle_{\top} \rangle_{\top}$ is satisfiable, where s is either a statement or an empty computation. It suffices to define and prove our relation \rightsquigarrow on such patterns, inductively on the structure of the first statement in s , if any:

- $\langle c_{pre} \langle \text{assert}(c_{post}) \rangle_k \rangle_{\top} \rightsquigarrow \cdot$ when $\models \langle \langle \cdot \rangle_k c_{pre} \rangle \Rightarrow \langle \langle \cdot \rangle_k c_{post} \rangle$
This is the case when s is empty. Note that this is the only way to rewrite the left-hand-side pattern using \rightarrow , too, and that $\overline{\Gamma}$ is indeed derivable whenever (actually “iff”) the condition holds.
- $\langle \langle x := e \rightsquigarrow s' \rightsquigarrow \text{assert}(c_{post}) \rangle_k \langle \rho \rangle_{\text{env}} c \rangle_{\top} \rightsquigarrow \langle \langle s' \rightsquigarrow \text{assert}(c_{post}) \rangle_k \langle \rho[\rho(e)/x] \rangle_{\text{env}} c \rangle_{\top}$

Note that the only way to rewrite the left-hand-side pattern above using \rightarrow is to first eventually transform e to $\rho(e)$ and then to apply the rewrite rule for assignment, so the relation \rightsquigarrow above is determined by the relation \rightarrow . Suppose that $\langle \langle s' \rightsquigarrow \text{assert}(c_{post}) \rangle_k \langle \rho[\rho(e)/x] \rangle_{\text{env}} c \rangle_{\top}$ is derivable. If s' is empty, then we have $\models \langle \langle \cdot \rangle_k \langle \rho[\rho(e)/x] \rangle_{\text{env}} c \rangle \Rightarrow \langle \langle \cdot \rangle_k c_{post} \rangle$ and $\langle \langle x := e \rightsquigarrow s' \rightsquigarrow \text{assert}(c_{post}) \rangle_k \langle \rho \rangle_{\text{env}} c \rangle_{\top}$ is the correctness

pair $\langle \langle \rho \rangle_{\text{env}} c \rangle_{\top} x := e \langle c_{post} \rangle_{\top}$, which is derivable by an application of (ML-ASGN) followed by an application of (ML-CONSEQUENCE). If s' is a statement, then we have that $\langle \langle \rho[\rho(e)/x] \rangle_{\text{env}} c \rangle_{\top} s' \langle c_{post} \rangle_{\top}$ is derivable and that $\langle \langle x := e \rightsquigarrow s' \rightsquigarrow \text{assert}(c_{post}) \rangle_k \langle \rho \rangle_{\text{env}} c \rangle_{\top}$ is the correctness pair $\langle \langle \rho \rangle_{\text{env}} c \rangle_{\top} x := e; s' \langle c_{post} \rangle_{\top}$, which is derivable by an application of (ML-ASGN) followed by an application of (ML-SEQ). Therefore, all the desired properties hold.

- $\langle \langle \text{if}(e) s_1 \text{ else } s_2 \rightsquigarrow s' \rightsquigarrow \text{assert}(c_{post}) \rangle_k \langle \varphi \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top} \rightsquigarrow \langle \langle s_1 \rightsquigarrow s' \rightsquigarrow \text{assert}(c_{post}) \rangle_k \langle \varphi \wedge \rho(e) \neq 0 \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top} \langle \langle s_2 \rightsquigarrow s' \rightsquigarrow \text{assert}(c_{post}) \rangle_k \langle \varphi \wedge \rho(e) = 0 \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top}$

First, note that the relation \rightsquigarrow is determined for the conditional, because once a conditional statement is on the top of some computation, the only thing which the rewrite system of \rightarrow can do with it is to eventually evaluate its condition e to $\rho(e)$ and then to apply the rule (V-IF) and generate two pattern tasks, like above. Before we prove this case, let us introduce a notion of “join pattern”. We show it for the particular configuration of IMP, but it works the same way for any configuration. If Γ_1 and Γ_2 are patterns over the same computation, say $\Gamma_1 = \langle \langle k \rangle_k \langle \rho_1 \rangle_{\text{env}} \langle X_1 \rangle_{\text{bnd}} \langle \varphi_1 \rangle_{\text{form}} \rangle$ and $\Gamma_2 = \langle \langle k \rangle_k \langle \rho_2 \rangle_{\text{env}} \langle X_2 \rangle_{\text{bnd}} \langle \varphi_2 \rangle_{\text{form}} \rangle$, then let $\Gamma_1 \nabla \Gamma_2$ be the *join pattern* $\langle \langle k \rangle_k \langle \rho \rangle_{\text{env}} \langle X_1 \cup X_2 \cup \{\rho\} \rangle_{\text{bnd}} \langle \varphi_1 \wedge \rho = \rho_1 \vee \varphi_2 \wedge \rho = \rho_2 \rangle_{\text{form}} \rangle$. Notice that $\models \Gamma_1 \nabla \Gamma_2 \Leftrightarrow \Gamma_1 \vee \Gamma_2$ but, unlike $\Gamma_1 \vee \Gamma_2$, $\Gamma_1 \nabla \Gamma_2$ is an actual pattern. Let us now assume that the correctness pairs corresponding to the two patterns in the right-hand-side of the rule above are derivable, i.e., $\langle \langle \varphi \wedge \rho(e) \neq 0 \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top} \overline{s_1}; \overline{s'} \langle c_{post} \rangle_{\top}$ and $\langle \langle \varphi \wedge \rho(e) = 0 \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top} \overline{s_2}; \overline{s'} \langle c_{post} \rangle_{\top}$ are derivable. That means that there must be some appropriate partial configurations c_1 and c_2 such that $\langle \langle \varphi \wedge \rho(e) \neq 0 \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top} \overline{s_1} \langle c_1 \rangle_{\top}$ and $\langle c_1 \rangle_{\top} \overline{s'} \langle c_{post} \rangle_{\top}$, and such that $\langle \langle \varphi \wedge \rho(e) = 0 \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top} \overline{s_2} \langle c_2 \rangle_{\top}$ and $\langle c_2 \rangle_{\top} \overline{s'} \langle c_{post} \rangle_{\top}$. Let Γ_1 be $\langle \langle \overline{s'} \rangle_k c_1 \rangle_{\top}$, let Γ_2 be $\langle \langle \overline{s'} \rangle_k c_2 \rangle_{\top}$, and let $\Gamma_1 \nabla \Gamma_2$ be the joint pattern constructed as above; by abuse of notation we let $c_1 \nabla c_2$ denote the partial configuration such that $\Gamma_1 \nabla \Gamma_2$ is $\langle \langle \overline{s'} \rangle_k c_1 \nabla c_2 \rangle_{\top}$. By (ML-CASES), $\langle c_1 \nabla c_2 \rangle_{\top} \overline{s'} \langle c_{post} \rangle_{\top}$ is derivable. By (ML-CONSEQUENCE) applied twice and (ML-IF) applied once, $\langle \langle \varphi \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top} \text{if}(e) s_1 \text{ else } s_2; \overline{s'} \langle c_{post} \rangle_{\top}$ is derivable, so $\langle \langle \varphi \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top} \text{if}(e) s_1 \text{ else } s_2; \overline{s'} \langle c_{post} \rangle_{\top}$ is derivable.

- $\langle \langle \text{while}(e) s \rightsquigarrow s' \rightsquigarrow \text{assert}(c_{post}) \rangle_k \langle \varphi \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top} \rightsquigarrow \langle \langle s \rightsquigarrow \text{assert}(c_{post}) \rangle_k \langle \varphi \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top} \langle \langle s' \rightsquigarrow \text{assert}(c_{post}) \rangle_k \langle \varphi \wedge \rho(e) = 0 \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top}$

First, note that the rule above for \rightsquigarrow is determined by \rightarrow . Indeed, the only thing \rightarrow can do with a while on top of the computation is to rewrite it to an if using rule (V-WHILE), then to process the condition e of the resulted conditional until it becomes $\rho(e)$, and then to apply the rule (V-IF) to split the conditional in the two cases appearing in the right-hand-side of the rule above. Suppose that $\langle \langle \varphi \wedge \rho(e) \neq 0 \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top} \overline{s} \langle \langle \varphi \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top}$ and $\langle \langle \varphi \wedge \rho(e) = 0 \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top} \overline{s} \langle c_{post} \rangle_{\top}$ are derivable. Then by (ML-WHILE) and (ML-SEQ), $\langle \langle \varphi \rangle_{\text{form}} \langle \rho \rangle_{\text{env}} c \rangle_{\top} \text{while}(e) s; \overline{s'} \langle c_{post} \rangle_{\top}$ is also derivable.

(Completeness) The idea of the proof of completeness is to fully annotate the program with assertions corresponding to all the pre- and post-conditions for all the fragments of code obtained from the matching logic proof, and then to show that the fully annotated program verifies. In order to prove the result modularly by induction, we need to distinguish paths that correspond to normally terminated programs from paths that correspond to loop body checks, because the latter do not get composed with the remaining program. Therefore, let us introduce a special statement “stop” that will be used to annotate the end of loop bodies; stop counts as an annotation, that is, it is simply discarded like any other annotation by the oper-

ation $s \mapsto \bar{s}$. Let ξ be a mapping from matching logic proofs into annotated programs defined recursively over proofs as follows:

- If π_{asgn} is the (trivial) matching logic derivation of the pair $\langle c_{\text{pre}} \rangle \mathbf{x} := e \langle c_{\text{post}} \rangle$, then let $\xi(\pi_{\text{asgn}})$ be the annotated program $\text{assert}\langle c_{\text{pre}} \rangle; \mathbf{x} := e; \text{assert}\langle c_{\text{post}} \rangle$;
- If π_{seq} is the derivation of $\langle c_{\text{pre}} \rangle s_1; s_2 \langle c_{\text{post}} \rangle$ and π_{seq_1} and π_{seq_2} are the corresponding derivations of $\langle c_{\text{pre}} \rangle s_1 \langle c \rangle$ and $\langle c \rangle s_2 \langle c_{\text{post}} \rangle$ (for the appropriate c), then let $\xi(\pi_{\text{seq}})$ be the annotated program $\text{assert}\langle c_{\text{pre}} \rangle; \xi(\pi_{\text{seq}_1}); \xi(\pi_{\text{seq}_2}); \text{assert}\langle c_{\text{post}} \rangle$;
- If π_{if} is a derivation of $\langle c_{\text{pre}} \rangle \text{if}(e) s_1 \text{ else } s_2 \langle c_{\text{post}} \rangle$ an π_{if_1} and π_{if_2} are the derivations of s_1 and s_2 corresponding to this particular proof (for the sake of saving space, we do not mention the exact configuration ingredients), then let $\xi(\pi_{\text{if}})$ be the annotated program $\text{assert}\langle c_{\text{pre}} \rangle; \text{if}(e) \xi(\pi_{\text{if}_1}) \text{ else } \xi(\pi_{\text{if}_2}); \text{assert}\langle c_{\text{post}} \rangle$;
- If π_{while} is a derivation of $\langle c_{\text{pre}} \rangle \text{while}(e) s \langle c_{\text{post}} \rangle$ and π_{body} is the derivation of s corresponding to this particular proof (for the sake of saving space, we do not mention the exact configuration ingredients), then let $\xi(\pi_{\text{while}})$ be the annotated program $\text{assert}\langle c_{\text{pre}} \rangle; \text{while}(e) (\xi(\pi_{\text{body}}); \text{stop}); \text{assert}\langle c_{\text{post}} \rangle$. Since the while is rewritten by the verifier into a conditional checking the invariant as an assertion after its positive branch corresponding to the loop body, we need to allow for switching the stop statement with a subsequent assertion, so that stop will indeed become the end of the computation. We therefore add the rule $\text{stop}; \text{assert}\langle c \rangle \rightarrow \text{assert}\langle c \rangle; \text{stop}$

It is easy to see that for any statement s for which a derivation $\langle c_{\text{pre}} \rangle s \langle c_{\text{post}} \rangle$ exists in matching logic, say π_s , it is indeed the case that $\xi(\pi_s) = s$. All we need to show is that $\langle \langle \xi(\pi_s) \rangle_k c_{\text{pre}} \rangle_{\top} \rightarrow^* \langle \Gamma_{\text{stop}} \rangle_{\top}$, where Γ_{stop} contains only patterns of the form $\langle \langle \text{stop} \rangle_k c \rangle$. We proceed by proving a stronger result by induction on s and its derivation, namely that $\langle \langle \xi(\pi_s) \rangle_k c_{\text{pre}} \rangle_{\top} \rightarrow^* \langle \Gamma_{\text{stop}} \langle \langle \cdot \rangle_k c_{\text{post}} \rangle \rangle_{\top}$. We discuss each of the four cases above corresponding to the each of the four types of statements:

- $\xi(\pi_{\text{asgn}}) = \text{assert}\langle c_{\text{pre}} \rangle; \mathbf{x} := e; \text{assert}\langle c_{\text{post}} \rangle$
We start with the term $\langle \langle \xi(\pi_{\text{asgn}}) \rangle_k c_{\text{pre}} \rangle_{\top}$. The first assertion is obviously discarded by rule (V-ASSERTION-CHECKING), because the current configuration is precisely the asserted assertion. Then the rewrite rule for assignment modifies the value of \mathbf{x} in the environment ρ of c_{pre} to $\rho(e)$, so the configuration before $\text{assert}\langle c_{\text{post}} \rangle$ is exactly c_{post} . Therefore rule (V-ASSERTION-CHECKING) applies again and rewrites the term to $\langle \langle \cdot \rangle_k c_{\text{post}} \rangle_{\top}$.
- $\xi(\pi_{\text{seq}}) = \text{assert}\langle c_{\text{pre}} \rangle; \xi(\pi_{\text{seq}_1}); \xi(\pi_{\text{seq}_2}); \text{assert}\langle c_{\text{post}} \rangle$
We start with the term $\langle \langle \xi(\pi_{\text{seq}}) \rangle_k c_{\text{pre}} \rangle_{\top}$ and, as above, the first assertion is immediately discarded by rule (V-ASSERTION-CHECKING). Assume, inductively, that the property holds for $\xi(\pi_{\text{seq}_1})$ and $\xi(\pi_{\text{seq}_2})$, i.e., $\langle \langle \xi(\pi_{\text{seq}_1}) \rangle_k c_{\text{pre}} \rangle_{\top} \rightarrow^* \langle \Gamma_{\text{stop}}^1 \langle \langle \cdot \rangle_k c \rangle \rangle_{\top}$ and $\langle \langle \xi(\pi_{\text{seq}_2}) \rangle_k c \rangle_{\top} \rightarrow^* \langle \Gamma_{\text{stop}}^2 \langle \langle \cdot \rangle_k c_{\text{post}} \rangle \rangle_{\top}$. Since all rules in Figure 6 only match and change the computations at the top of the $\langle \dots \rangle_k$ cell and since the while loops generate a top-level conditional whose positive branch discards the computation that follows the while loop, the computations in the left terms and the empty computations in the right terms in the rewrites above can be appended any other computations and the rewrites would still hold, in particular $\langle \langle \xi(\pi_{\text{seq}_1}); \xi(\pi_{\text{seq}_2}); \text{assert}\langle c_{\text{post}} \rangle \rangle_k c_{\text{pre}} \rangle_{\top} \rightarrow^* \langle \Gamma_{\text{stop}}^1 \langle \langle \xi(\pi_{\text{seq}_2}); \text{assert}\langle c_{\text{post}} \rangle \rangle_k c \rangle \rangle_{\top}$ and $\langle \langle \xi(\pi_{\text{seq}_2}); \text{assert}\langle c_{\text{post}} \rangle \rangle_k c \rangle_{\top} \rightarrow^* \langle \Gamma_{\text{stop}}^2 \langle \langle \text{assert}\langle c_{\text{post}} \rangle \rangle_k c_{\text{post}} \rangle \rangle_{\top}$. Since the patterns in Γ_{stop}^1 do not interfere with the rewrites applied on $\langle \langle \xi(\pi_{\text{seq}_2}); \text{assert}\langle c_{\text{post}} \rangle \rangle_k c \rangle_{\top}$, the two rules above

can apply one after the other, then followed by an instance of (V-ASSERTION-CHECKING), and thus we obtain $\langle \langle \xi(\pi_{\text{seq}_1}); \xi(\pi_{\text{seq}_2}); \text{assert}\langle c_{\text{post}} \rangle \rangle_k c_{\text{pre}} \rangle_{\top} \rightarrow^* \langle \Gamma_{\text{stop}}^1 \Gamma_{\text{stop}}^2 \langle \langle \cdot \rangle_k c_{\text{post}} \rangle \rangle_{\top}$.

- $\xi(\pi_{\text{if}}) = \text{assert}\langle c_{\text{pre}} \rangle; \text{if}(e) \xi(\pi_{\text{if}_1}) \text{ else } \xi(\pi_{\text{if}_2}); \text{assert}\langle c_{\text{post}} \rangle$
We start with the term $\langle \langle \xi(\pi_{\text{if}}) \rangle_k c_{\text{pre}} \rangle_{\top}$ and, as above, the first assertion is discarded by rule (V-ASSERTION-CHECKING). Then the only rewrite steps which can take place are those that lead to the evaluation of e to $\rho(e)$, where ρ is the environment of c_{pre} . Once e is evaluated, the only rule which can take place is (V-IF), yielding $\langle \langle \xi(\pi_{\text{if}_1}) \rangle_k c_{\text{pre}} \wedge \rho(e) \neq 0 \rangle \langle \langle \xi(\pi_{\text{if}_2}) \rangle_k c_{\text{pre}} \wedge \rho(e) = 0 \rangle_{\top}$. We can now apply the induction hypothesis and obtain that $\langle \langle \xi(\pi_{\text{if}_1}) \rangle_k c_{\text{pre}} \wedge \rho(e) \neq 0 \rangle_{\top} \rightarrow^* \langle \Gamma_{\text{stop}}^1 \langle \langle \cdot \rangle_k c_{\text{post}} \rangle \rangle_{\top}$ and that $\langle \langle \xi(\pi_{\text{if}_2}) \rangle_k c_{\text{pre}} \wedge \rho(e) = 0 \rangle_{\top} \rightarrow^* \langle \Gamma_{\text{stop}}^2 \langle \langle \cdot \rangle_k c_{\text{post}} \rangle \rangle_{\top}$. Therefore, $\langle \langle \xi(\pi_{\text{if}}) \rangle_k c_{\text{pre}} \rangle_{\top} \rightarrow^* \langle \Gamma_{\text{stop}}^1 \Gamma_{\text{stop}}^2 \langle \langle \cdot \rangle_k c_{\text{post}} \rangle \rangle_{\top}$.
- $\xi(\pi_{\text{while}}) = \text{assert}\langle c_{\text{pre}} \rangle; \text{while}(e) (\xi(\pi_{\text{body}}); \text{stop}); \text{assert}\langle c_{\text{post}} \rangle$
We start with the term $\langle \langle \xi(\pi_{\text{while}}) \rangle_k c_{\text{pre}} \rangle_{\top}$ and, as above, the first assertion is discarded by rule (V-ASSERTION-CHECKING). The rule (V-WHILE) can be applied, yielding the term $\langle \langle \text{if}(e) (\xi(\pi_{\text{body}}); \text{stop}); \text{assert}\langle c_{\text{pre}} \rangle \rangle_k c_{\text{pre}} \rangle_{\top}$ which further rewrites into $\langle \langle \xi(\pi_{\text{body}}); \text{assert}\langle c_{\text{pre}} \rangle; \text{stop} \rangle_k c_{\text{pre}} \wedge \rho(e) \neq 0 \rangle_{\top} \langle \langle \text{assert}\langle c_{\text{post}} \rangle \rangle_k c_{\text{pre}} \wedge \rho(e) = 0 \rangle_{\top}$. However, in this case c_{post} is precisely $c_{\text{pre}} \wedge \rho(e) = 0$, so the second pattern rewrites to $\langle \langle \cdot \rangle_k c_{\text{post}} \rangle_{\top}$. The induction hypothesis for $\xi(\pi_{\text{body}})$ gives us the following rewriting sequence: $\langle \langle \xi(\pi_{\text{body}}) \rangle_k c_{\text{pre}} \wedge \rho(e) \neq 0 \rangle_{\top} \rightarrow^* \langle \Gamma_{\text{stop}} \langle \langle \cdot \rangle_k c_{\text{pre}} \rangle \rangle_{\top}$. Then we can deduce that $\langle \langle \xi(\pi_{\text{body}}); \text{assert}\langle c_{\text{pre}} \rangle; \text{stop} \rangle_k c_{\text{pre}} \wedge \rho(e) \neq 0 \rangle_{\top} \rightarrow^* \langle \Gamma_{\text{stop}} \langle \langle \text{assert}\langle c_{\text{pre}} \rangle; \text{stop} \rangle_k c_{\text{pre}} \rangle \rangle_{\top} \rightarrow \langle \Gamma_{\text{stop}} \langle \langle \text{stop} \rangle_k c_{\text{pre}} \rangle \rangle_{\top}$. Thus, we have proved that $\langle \langle \xi(\pi_{\text{while}}) \rangle_k c_{\text{pre}} \rangle_{\top} \rightarrow^* \langle \Gamma_{\text{stop}} \langle \langle \text{stop} \rangle_k c_{\text{pre}} \rangle \langle \langle \cdot \rangle_k c_{\text{post}} \rangle \rangle_{\top}$.

The completeness proof is done. Note that we generated many more annotations than needed with the translation ξ . Since our verifier is capable of performing many steps without a need for annotations, it would be interesting to research ways to produce minimal annotations in verified programs so that those can be checked without any involved search. However, that was not our purpose here. THEOREM 12

Thus, although in principle one needs to provide pre- and post-condition patterns for all statements in a program, the verifier discussed above has the advantage that it calculates many of those patterns automatically. Indeed, given a pre-condition pattern, it uses symbolically the actual executable semantics of the current language construct to calculate the post-condition pattern, at the same time generating proof obligations for SMT solvers whenever given pattern assertions are reached during the rewriting process. In our experience with an implementation of a matching logic prover in Maude (see Section 7), in addition to pre- and post-condition patterns for the program to verify one typically only needs to provide pattern annotations for loop invariants.

One potential practical issue with our verification approach above is the so-called path explosion problem. First, note that since in term rewriting one can choose any rule that matches to apply, our \mathbb{K} verifier definition has several degrees of freedom w.r.t. how it can be implemented. For example, one can discard verification tasks as soon as their condition becomes false (fourth rule in step (3) in Figure 6), thus avoiding exploring infeasible paths. Second, one can use pattern assertions anywhere in the program, in particular right after conditionals. If that is the case, then the two verification tasks corresponding to the two branches become identical right after the

$S ::= \dots$
$\quad \quad PVar := \text{cons}(\text{Seq}^+ [E])$
$\quad \quad PVar := [E]$
$\quad \quad [E_1] := E_2$
$\quad \quad \text{dispose}(E)$
$\text{CfgItem} ::= \dots \langle \text{Mem} \rangle_{\text{mem}}$
$\text{Mem} ::= \text{Map}^{\otimes} [\text{Nat}^+, \text{Int}]$
$(x := \text{cons}(\bar{i}, e, \bar{e})) = (e \leadsto x := \text{cons}(\bar{i}, \square, \bar{e}))$
$(x := [e]) = (e \leadsto x := [\square])$
$([e_1] := e_2) = (e_1 \leadsto [\square] := e_2) \quad ([p] := e_2) = (e_2 \leadsto [p] := \square)$
$\text{dispose}(e) = (e \leadsto \text{dispose}(\square))$
$\langle x := \text{cons}(\bar{i}, e, \bar{e}) \rangle_k \langle \rho \rangle_{\text{env}} \langle \sigma \rangle_{\text{mem}} \rightarrow \langle k \rangle_k \langle \rho[p/x] \rangle_{\text{env}} \langle p \mapsto [\bar{i}] \otimes \sigma \rangle_{\text{mem}}$
$\langle x := [p] \rangle_k \langle \rho \rangle_{\text{env}} \langle p \mapsto i \otimes \sigma \rangle_{\text{mem}} \rightarrow \langle k \rangle_k \langle \rho[i/x] \rangle_{\text{env}} \langle p \mapsto i \otimes \sigma \rangle_{\text{mem}}$
$\langle [p] := i \rangle_k \langle \rho \mapsto i' \otimes \sigma \rangle_{\text{mem}} \rightarrow \langle k \rangle_k \langle \rho \mapsto i \otimes \sigma \rangle_{\text{mem}}$
$\langle \text{dispose}(p) \rangle_k \langle \rho \mapsto i \otimes \sigma \rangle_{\text{mem}} \rightarrow \langle k \rangle_k \langle \sigma \rangle_{\text{mem}}$

(range of new variables: $p \in \text{Nat}^+$; $\bar{i} \in \text{Seq}^+ [\text{Int}]$; $\bar{e} \in \text{Seq}^+ [E]$)

Figure 7. HIMP in \mathbb{K} (features above added to those in Figure 2)

pattern assertion at the end of the conditional is processed (recall that the top level cell is a set, so duplicates are eliminated); this way, the path explosion problem is avoided.

6. Adding a Heap

We next define HIMP (IMP with *heap*), an extension of IMP with dynamic memory allocation/deallocation and pointer arithmetic. We show that the three definitions related to IMP presented so far (i.e., its \mathbb{K} definition, its matching logic formal system, and its \mathbb{K} matching logic verifier) extend modularly to HIMP; we also show the corresponding correctness theorems. The heap allows for introducing and axiomatizing heap data-structures by means of pointers, such as lists, trees, graphs, etc. We define some of these and prove the list reverse as an example (Section 7 discusses a more elaborate case study, the Schorr-Waite algorithm). It is worth mentioning that, unlike in separation logic where such data-structures are defined by means of recursive predicates, we define them as ordinary term constructs, with natural (purely first-order) axioms saying how they can be identified and/or manipulated in configurations or patterns.

6.1 \mathbb{K} Definition of HIMP

Figure 7 shows the features that need to be added to the \mathbb{K} definition of IMP to obtain an executable definition of HIMP. We follow the simple syntax and semantics proposed by Reynolds [2002] for memory operations, namely: statement $x := \text{cons}(e_1, \dots, e_n)$ evaluates e_1, \dots, e_n to values i_1, \dots, i_n , allocates a contiguous block of size n available starting with pointer (a positive integer) p in memory and writes values i_1, \dots, i_n in order in that block, and finally assigns p to x ; $x := [e]$ evaluates e to a pointer p which must be allocated, and assigns to x the value at location p ; $[e_1] := e_2$ evaluates e_1 to pointer p which must be allocated and e_2 to value i , and writes i to location p ; $\text{dispose}(e)$ evaluates e to a pointer p (which must have been allocated), and deallocates the location p . Note how we define the strictness of cons , a style also supported by our tool (see Section 7): \bar{i} ranges over (possibly empty) sequences of values, while \bar{e} ranges over (possibly empty) sequences of expressions; this way, the term $x := \text{cons}(\bar{i}, e, \bar{e})$ matches the first non-value argument expression e . This equation applies iteratively to schedule all arguments of cons for processing, and to plug back their results into the cons after they are processed.

To define the semantics of HIMP, we need to first extend the configuration of IMP with a *heap*, or *memory* cell. A heap is a (partial) map structure just like the environment, but from positive naturals (also called pointers) to integers. We use \otimes as a heap construct to make our heap notation resemble that of separation logic, but note that there is big difference between the two: our \otimes is a term constructor, while $*$ in separation logic is a special logical connective. Note that, thanks to \mathbb{K} 's modularity, adding a new cell to the configuration does not affect the existing semantic equations and rules (e.g., those of IMP in Figure 2) at all, because they simply match and use/modify only what they need from the configuration. Therefore, we only add one rule for each new construct. Due to the strictness equations, we can assume that all the arguments have been evaluated. The semantics of cons reads as follows: if “ $x := \text{cons}(\bar{i})$ ” is the next computational task under environment ρ and memory σ , then pick some arbitrary pointer p such that $p \mapsto [\bar{i}] \otimes \sigma$ is a well-formed map, discard the statement and update the environment and the memory accordingly; here and elsewhere in the paper we use the notation “ $p \mapsto [i_1, i_2, \dots, i_n]$ ” as a shorthand for “ $p \mapsto i_1 \otimes p+1 \mapsto i_2 \otimes \dots \otimes p+n-1 \mapsto i_n$ ”, inspired from separation logic. The semantic rule of cons above is compact, elegant and natural, but it is rather dense. There are at least two important aspects of it that need to be noticed. (1) How can one choose such a p ? Rewriting logic allows variables that do not appear in the left-hand-sides of rules to appear in the right-hand-sides, like our p , with the semantics that such a rule can be regarded as infinitely many instances, one for each instance of that variable; that is why we called p an “arbitrary pointer” above. (2) How can we make sure that the allocated memory was available? All the terms appearing in a rule are assumed to have the expected sorts, satisfying all the corresponding sort constraints. In our case, the term $p \mapsto [\bar{i}] \otimes \sigma$ appears where a map sort is expected, so it must be a well-formed map. Thinking in terms of “rule schemata” (standing for the recursively enumerable set of its ground instances), the rule of cons can be regarded as an infinite set of ground rules, one for each choice of p satisfying all the sort constraints in the rule. The semantics of the remaining three rules should be clear now.

We adopt and extend in the obvious way all the notions introduced in Definition 1 for IMP. In particular, $\llbracket s \rrbracket$ denotes the initial configuration $\langle \langle s \rangle_k \langle \cdot \rangle_{\text{env}} \langle \cdot \rangle_{\text{mem}} \rangle$. We next introduce a HIMP specific notion, namely memory safety. Note that the rewriting process using the HIMP semantic rules in Figure 7 will get stuck when pointer p is not available in the heap (or, in the case of the cons rule, when the block cannot be allocated — due to memory bounds constraints, if any). This leads to a natural definition of memory safety:

DEFINITION 13. HIMP configuration γ is **memory safe** iff it is not the case that $\text{HIMP} \models \gamma \rightarrow^* \gamma'$ for some junk configuration γ' which is stuck with a memory access construct at the beginning of its computation (in its $\langle \dots \rangle_k$ cell).

To execute the \mathbb{K} semantics in Figure 7, one needs to make a concrete choice for the pointer p in the rule for cons , that is, one needs to define a “memory manager”. Roşu et al. [2009] show how to give executable \mathbb{K} semantics to C-like languages where p is chosen symbolically; this makes the language definition deterministic and leads to a strictly stronger notion of memory safety.

We prove a similar result to PROPOSITION 2, stating that tasks in the computation structure are processed in order.

PROPOSITION 14. Given $k \in E \cup S$ and $r \in K$, then a rewriting sequence $\langle \langle k \sim r \rangle_k \langle \rho \rangle_{\text{env}} \langle \sigma \rangle_{\text{mem}} \rangle \rightarrow^* \gamma$ is possible for some final configuration γ iff there is some final configuration $\gamma' = \langle \langle k' \rangle_k \langle \rho' \rangle_{\text{env}} \langle \sigma' \rangle_{\text{mem}} \rangle$ such that $\langle \langle k \rangle_k \langle \rho \rangle_{\text{env}} \langle \sigma \rangle_{\text{mem}} \rangle \rightarrow^* \gamma'$ and $\langle \langle k' \rangle_k \langle \rho' \rangle_{\text{env}} \langle \sigma' \rangle_{\text{mem}} \rangle \rightarrow^* \gamma$; moreover, if that is the case then $k' = \rho(k)$, $\rho' = \rho$ and $\sigma' = \sigma$ when $k \in E$, and $k' = \cdot$ when $k \in S$.

$$\begin{array}{c}
\frac{C[\bar{e}] = \bar{v} \text{ with } C = \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \langle X \rangle_{bnd} C'}{\langle C \rangle x := \text{cons}(\bar{e}) \langle \rho[p/x] \rangle_{env} \langle p \mapsto [\bar{v}] \otimes \sigma \rangle_{mem} \langle X, p \rangle_{bnd} C'} \quad (\text{ML-CONS}) \\
\\
\frac{C[e] \equiv v, \text{ with } C = \langle \rho \rangle_{env} \langle v \mapsto v' \otimes \sigma \rangle_{mem} C'}{\langle C \rangle x := [e] \langle \rho[v'/x] \rangle_{env} \langle v \mapsto v' \otimes \sigma \rangle_{mem} C'} \quad (\text{ML-ASGN}) \\
\\
\frac{C[(e_1, e_2)] \equiv (v_1, v_2), \text{ with } C = \langle v_1 \mapsto v'_2 \otimes \sigma \rangle_{mem} C'}{\langle C \rangle [e_1] := [e_2] \langle v_1 \mapsto v_2 \otimes \sigma \rangle_{mem} C'} \quad (\text{ML-LOOKUP}) \\
\\
\frac{C[e] \equiv v, \text{ with } C = \langle v \mapsto v' \otimes \sigma \rangle_{mem} C'}{\langle C \rangle \text{dispose}(e) \langle \sigma \rangle_{mem} C'} \quad (\text{ML-DISPOSE})
\end{array}$$

Figure 8. Matching logic formal system for HIMP (the rules above are added to those in Figure 4 — rules in Figure 4 stay unchanged)

Proof. The proof of “if” part is identical to that of PROPOSITION 2 and does not depend upon the particular language constructs. The “only if” part goes by structural induction on k and it is identical to that of PROPOSITION 2 for the IMP constructs, so we only discuss the new constructs.

Let us now analyze the cases where $k \in S$.

If $k = (x := \text{cons}(\bar{e}))$ then by the induction hypothesis applied on each expression in \bar{e} , it follows that $\langle k \leadsto r \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rightarrow^* \langle x := \text{cons}(\rho(\bar{e})) \leadsto r \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rightarrow^* \gamma$; the only way to advance the rewriting of $\langle x := \text{cons}(\rho(\bar{e})) \leadsto r \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem}$ is to apply the semantic rewrite rule for cons, so we obtain that $\langle x := \text{cons}(\rho(\bar{e})) \leadsto r \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rightarrow \langle k \rangle_k \langle \rho[p/x] \rangle_{env} \langle p \mapsto [\rho(\bar{e})] \otimes \sigma \rangle_{mem} \rightarrow^* \gamma$ for some arbitrary p . Then pick $k' = \cdot$, $\rho' = \rho[p/x]$ and $\sigma' = p \mapsto [\rho(\bar{e})] \otimes \sigma$ and note that the property holds.

If $k = (x := [e])$ then by the induction hypothesis on e , we get $\langle k \leadsto r \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rightarrow^* \langle x := [\rho(e)] \leadsto r \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rightarrow^* \gamma$; the only way for the latter rewrite sequence to exist is that the location $\rho(e)$ exists in σ , that is $\sigma = \rho(e) \mapsto v \otimes \sigma_0$, and an instance of the semantic rule of location lookup is applied, that is, $\langle x := [\rho(e)] \leadsto r \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rightarrow \langle k \rangle_k \langle \rho[v/x] \rangle_{env} \langle \sigma \rangle_{mem} \rightarrow^* \gamma$. Then pick $k' = \cdot$, $\rho' = \rho[v/x]$ and $\sigma' = \sigma$.

If $k = ([e_1] := e_2)$ then by the induction hypothesis applied first on e_1 and then on e_2 , we get $\langle k \leadsto r \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rightarrow^* \langle [e_1] := \rho(e_2) \leadsto r \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rightarrow^* \gamma$; the only way for the latter rewrite sequence to exist is that the location $\rho(e_1)$ exists in σ , that is $\sigma = \rho(e_1) \mapsto v \otimes \sigma_0$, and the semantic rule of location update is applied, that is, $\langle [e_1] := \rho(e_2) \leadsto r \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rightarrow \langle k \rangle_k \langle \rho \rangle_{env} \langle \rho(e_1) \mapsto \rho(e_2) \otimes \sigma_0 \rangle_{mem} \rightarrow^* \gamma$. Then pick $k' = \cdot$, $\rho' = \rho$ and $\sigma' = \rho(e_1) \mapsto \rho(e_2) \otimes \sigma_0$ and note that the property holds.

If $k = \text{dispose}(e)$ then by the induction hypothesis on e , we get $\langle k \leadsto r \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rightarrow^* \langle \text{dispose}(\rho(e)) \leadsto r \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rightarrow^* \gamma$; the only way for the latter rewrite sequence to exist is that the location $\rho(e)$ exists in σ , that is $\sigma = \rho(e) \mapsto v \otimes \sigma_0$, and an instance of the semantic rule of dispose is applied, that is, $\langle \text{dispose}(\rho(e)) \leadsto r \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rightarrow \langle k \rangle_k \langle \rho \rangle_{env} \langle \sigma_0 \rangle_{mem} \rightarrow^* \gamma$. Then pick $k' = \cdot$, $\rho' = \rho$ and $\sigma' = \sigma_0$. PROPOSITION 14

6.2 Matching Logic Definition of HIMP

Figure 8 shows the rules that need to be added to the formal system of IMP in Figure 4 to obtain a matching logic formal system for HIMP. In order for these rules to work, we extend IMP patterns to correspond to configurations extended with a heap, as discussed in Section 6.1. Since patterns inherit the structure of configurations, matching logic is as modular as \mathbb{K} . In particular, none of the matching logic rules in Figure 4 need to change when we add a heap to the configuration! To obtain a matching logic semantics for HIMP, all we need to add is one rule for each new language construct, as shown in Figure 8. To save space, we write

“ $C[(e_1, e_2, \dots, e_n)] \equiv (v_1, v_2, \dots, v_n)$ ” as a shorthand for “ $C[e_1] \equiv v_1$ and $C[e_2] \equiv v_2$ and ... and $C[e_n] \equiv v_n$ ”.

The rules in Figure 8 are self-explanatory, mechanically following their \mathbb{K} counterparts in Figure 7. The only interesting thing to note is that the “arbitrary pointer” in the \mathbb{K} rule of cons now gets existentially quantified in the pattern, so it becomes a pattern bound parameter. This way, the matching logic proof system becomes deterministic and, as expected, when a program is verified using matching logic then it is necessarily memory safe.

THEOREM 15. (Soundness of matching logic formal system for HIMP) Suppose that $\Gamma \Downarrow \Gamma'$ is derivable with the proof system in Figure 8 and that $(\gamma, \tau) : \text{Var}^\circ \rightarrow \mathcal{T}$ is such that $(\gamma, \tau) \models \Gamma$. Then:

1. γ is memory safe; and
2. If $\text{HIMP} \models \gamma \rightarrow^* \gamma'$ with γ' final then $(\gamma', \tau) \models \Gamma'$.

Proof. We do the proof in the same style as the proof of Theorem 8. As already mentioned in the proof of Theorem 8, there we never used the fact that the configurations of IMP had only two cells, a computation and an environment. In other words, the same proof of Theorem 8 can be used here for the soundness of the IMP constructs in HIMP, so we only need to continue with the soundness of the new constructs. Note, however, that in Theorem 8 we did not need to worry about memory safety, but now we do. None of the operations used on the proof of Theorem 8 affected in any way memory safety, in that the memory safety of the configurations satisfying the correctness pairs in the hypotheses of rules ensures the memory safety of the configuration satisfying the derived correctness pair. In other words, the memory safety is also preserved inductively over the rules proved sound in Theorem 8.

Like in the proof of Theorem 8, we consider the rules in Figure 8 in desugared form. For each of the rules deriving a sequent of the form $\langle C \rangle s \langle C' \rangle$, desugared to $\exists X. (o = \langle \langle s \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rangle \wedge \varphi) \Downarrow \exists X'. (o = \langle \langle \cdot \rangle_k \langle \rho' \rangle_{env} \langle \sigma' \rangle_{mem} \rangle \wedge \varphi')$, consider that $(\gamma, \tau) \models \exists X. (o = \langle \langle s \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rangle \wedge \varphi)$, that is, that γ is the term $\langle \langle s \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \langle \theta_\tau(\sigma) \rangle_{mem} \rangle$ for some $\theta_\tau : \text{Var} \rightarrow \mathcal{T}$ s.t. $\theta_\tau \upharpoonright_{\text{Var} \setminus X} = \tau \upharpoonright_{\text{Var} \setminus X}$ and $\theta_\tau \models \varphi$.

(ML-CONS)

$$\frac{\exists X. (o = \langle \langle \bar{e} \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rangle \wedge \varphi) \Downarrow \exists X. (o = \langle \langle \bar{v} \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rangle \wedge \varphi)}{\exists X. (o = \langle \langle x := \text{cons}(\bar{e}) \rangle_k \langle \rho \rangle_{env} \langle \sigma \rangle_{mem} \rangle \wedge \varphi) \Downarrow \exists X. p. (o = \langle \langle \cdot \rangle_k \langle \rho[p/x] \rangle_{env} \langle p \mapsto [\bar{v}] \otimes \sigma \rangle_{mem} \rangle \wedge \varphi)}$$

In this case γ has the form $\langle \langle x := \text{cons}(\bar{e}) \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \langle \theta_\tau(\sigma) \rangle_{mem} \rangle$, so by PROPOSITION 14 and LEMMA 11 γ can only rewrite (in possibly several steps) to the term $\langle \langle x := \text{cons}(\theta_\tau(\bar{v})) \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \langle \theta_\tau(\sigma) \rangle_{mem} \rangle$. The only semantic rewrite rule which can apply now is the one for cons, which rewrites the term above to a term of the form $\langle \langle \cdot \rangle_k \langle \theta_\tau(\rho)[p/x] \rangle_{env} \langle p \mapsto \theta_\tau(\bar{v}) \otimes \sigma \rangle_{mem} \rangle$, for some arbitrary natural number (or pointer) p , which is final. Therefore, γ is memory safe and, moreover, any final γ' with $\text{HIMP} \models \gamma \rightarrow^* \gamma'$ must have the same form. It is easy to see now that $(\gamma', \tau) \models \exists X. p. (o = \langle \langle \cdot \rangle_k \langle \rho[p/x] \rangle_{env} \langle p \mapsto [\bar{v}] \otimes \sigma \rangle_{mem} \rangle \wedge \varphi)$ (pick θ'_τ defined like θ_τ modified only in p , where $\theta'_\tau(p) = p$).

(ML-ASGN)

$$\frac{\exists X. (o = \langle \langle e \rangle_k \langle \rho \rangle_{env} \langle v \mapsto v' \otimes \sigma \rangle_{mem} \rangle \wedge \varphi) \Downarrow \exists X. (o = \langle \langle v \rangle_k \langle \rho \rangle_{env} \langle v \mapsto v' \otimes \sigma \rangle_{mem} \rangle \wedge \varphi)}{\exists X. (o = \langle \langle x := [e] \rangle_k \langle \rho \rangle_{env} \langle v \mapsto v' \otimes \sigma \rangle_{mem} \rangle \wedge \varphi) \Downarrow \exists X. (o = \langle \langle \cdot \rangle_k \langle \rho[v'/x] \rangle_{env} \langle v \mapsto v' \otimes \sigma \rangle_{mem} \rangle \wedge \varphi)}$$

In this case γ has the form $\langle \langle x := [e] \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \langle \theta_\tau(v \mapsto v' \otimes \sigma) \rangle_{mem} \rangle$, so by PROPOSITION 14 and LEMMA 11 γ can only rewrite (in possibly several steps) to $\langle \langle x := [\theta_\tau(v)] \rangle_k \langle \theta_\tau(\rho) \rangle_{env} \langle \theta_\tau(v \mapsto v' \otimes \sigma) \rangle_{mem} \rangle$. The

only rewrite rule which can apply is the one for pointer lookup, which yields $\langle\langle\cdot\rangle_k \langle\theta_\tau(\rho)[\theta_\tau(v')/\mathbf{x}]\rangle_{env} \langle\theta_\tau(v \mapsto v' \otimes \sigma)\rangle_{mem}\rangle$, which is final. Therefore, γ is memory safe and, moreover, the above is the only final γ' with $HIMP \models \gamma \rightarrow^* \gamma'$. It is easy to see now that $(\gamma', \tau) \models \exists X.(\circ = \langle\langle\cdot\rangle_k \langle\rho[v'/\mathbf{x}]\rangle_{env} \langle v \mapsto v' \otimes \sigma \rangle_{mem}\rangle \wedge \varphi)$.

(ML-[LOOKUP])

$$\begin{array}{c} \exists X.(\circ = \langle\langle\mathbf{e}_1\rangle_k \langle\rho\rangle_{env} \langle v_1 \mapsto v'_2 \otimes \sigma \rangle_{mem}\rangle \wedge \varphi) \\ \Downarrow \exists X.(\circ = \langle\langle v_1 \rangle_k \langle\rho\rangle_{env} \langle v_1 \mapsto v'_2 \otimes \sigma \rangle_{mem}\rangle \wedge \varphi), \\ \exists X.(\circ = \langle\langle\mathbf{e}_2\rangle_k \langle\rho\rangle_{env} \langle v_1 \mapsto v'_2 \otimes \sigma \rangle_{mem}\rangle \wedge \varphi) \\ \Downarrow \exists X.(\circ = \langle\langle v_2 \rangle_k \langle\rho\rangle_{env} \langle v_1 \mapsto v'_2 \otimes \sigma \rangle_{mem}\rangle \wedge \varphi) \\ \hline \exists X.(\circ = \langle\langle\mathbf{e}_1\rangle := \mathbf{e}_2\rangle_k \langle\rho\rangle_{env} \langle v_1 \mapsto v'_2 \otimes \sigma \rangle_{mem}\rangle \wedge \varphi) \\ \Downarrow \exists X.(\circ = \langle\langle\cdot\rangle_k \langle\rho\rangle_{env} \langle v_1 \mapsto v_2 \otimes \sigma \rangle_{mem}\rangle \wedge \varphi) \end{array}$$

In this case γ is $\langle\langle\mathbf{e}_1\rangle := \mathbf{e}_2\rangle_k \langle\theta_\tau(\rho)\rangle_{env} \langle\theta_\tau(v_1 \mapsto v'_2 \otimes \sigma)\rangle_{mem}$, so by PROPOSITION 14 and LEMMA 11 γ can only rewrite (in possibly several steps) to $\langle\langle\theta_\tau(v_1)\rangle := \theta_\tau(v_2)\rangle_k \langle\theta_\tau(\rho)\rangle_{env} \langle\theta_\tau(v_1 \mapsto v'_2 \otimes \sigma)\rangle_{mem}$. The only rewrite rule which can apply is the one for pointer assignment, which yields $\langle\langle\cdot\rangle_k \langle\theta_\tau(\rho)\rangle_{env} \langle\theta_\tau(v_1 \mapsto v_2 \otimes \sigma)\rangle_{mem}\rangle$, which is final. Therefore, γ is memory safe and, moreover, the above is the only final γ' with $HIMP \models \gamma \rightarrow^* \gamma'$. It is easy to see now that $(\gamma', \tau) \models \exists X.(\circ = \langle\langle\cdot\rangle_k \langle\rho\rangle_{env} \langle v_1 \mapsto v_2 \otimes \sigma \rangle_{mem}\rangle \wedge \varphi)$.

(ML-DISPOSE)

$$\begin{array}{c} \exists X.(\circ = \langle\langle\mathbf{e}\rangle_k \langle\rho\rangle_{env} \langle v \mapsto v' \otimes \sigma \rangle_{mem}\rangle \wedge \varphi) \\ \Downarrow \exists X.(\circ = \langle\langle v \rangle_k \langle\rho\rangle_{env} \langle v \mapsto v' \otimes \sigma \rangle_{mem}\rangle \wedge \varphi) \\ \hline \exists X.(\circ = \langle\langle\mathbf{dispose}(\mathbf{e})\rangle_k \langle\rho\rangle_{env} \langle v \mapsto v' \otimes \sigma \rangle_{mem}\rangle \wedge \varphi) \\ \Downarrow \exists X.(\circ = \langle\langle\cdot\rangle_k \langle\rho\rangle_{env} \langle\sigma\rangle_{mem}\rangle \wedge \varphi) \end{array}$$

In this case γ is $\langle\langle\mathbf{dispose}(\mathbf{e})\rangle_k \langle\theta_\tau(\rho)\rangle_{env} \langle\theta_\tau(v \mapsto v' \otimes \sigma)\rangle_{mem}\rangle$, so by PROPOSITION 14 and LEMMA 11 γ can only rewrite (in possibly several steps) to $\langle\langle\mathbf{dispose}(\theta_\tau(v))\rangle_k \langle\theta_\tau(\rho)\rangle_{env} \langle\theta_\tau(v \mapsto v' \otimes \sigma)\rangle_{mem}\rangle$. The only rewrite rule which can apply is the one for **dispose**, which yields $\langle\langle\cdot\rangle_k \langle\theta_\tau(\rho)\rangle_{env} \langle\sigma\rangle_{mem}\rangle$, which is final. Therefore, γ is memory safe and, moreover, the above is the only final γ' with $HIMP \models \gamma \rightarrow^* \gamma'$. It is easy to see now that $(\gamma', \tau) \models \exists X.(\circ = \langle\langle\cdot\rangle_k \langle\rho\rangle_{env} \langle\sigma\rangle_{mem}\rangle \wedge \varphi)$. THEOREM 15

6.3 Deriving a HIMP Matching Logic Verifier in \mathbb{K}

The \mathbb{K} matching logic verifier for IMP in Section 5 was obtained by adding configuration infrastructure for expressing patterns to the \mathbb{K} definition of IMP, four language independent rules, and modifying two of the original \mathbb{K} rules (for **if** and **while**) to become symbolic. We follow the same steps for HIMP and, surprisingly, we only have to modify the rule for **cons**. Concretely, the matching logic verifier for HIMP is obtained by extending the one for IMP in Figure 6 with HIMP configurations (i.e., adding memory cell) and with the HIMP \mathbb{K} rules in Figure 7 modifying the one for **cons** as follows:

$$\begin{array}{c} \langle\mathbf{x} := \mathbf{cons}(\bar{\mathbf{i}}) \rangle \sim \langle\mathbf{k}\rangle_k \langle\rho\rangle_{env} \langle\sigma\rangle_{mem} \langle X \rangle_{bnd} \\ \rightarrow \langle\mathbf{k}\rangle_k \langle\rho[\mathbf{p}/\mathbf{x}]\rangle_{env} \langle\mathbf{p} \mapsto [\bar{\mathbf{i}}] \otimes \sigma\rangle_{mem} \langle X, \mathbf{p} \rangle_{bnd} \end{array}$$

Therefore, the pointer \mathbf{p} of the original \mathbb{K} semantics is now bound by the right-hand-side pattern, making it a symbolic value for the remainder of the proof. The soundness and completeness of the HIMP verifier obtained as above w.r.t. the matching logic proof system in Figure 8 is similar to THEOREM 12, so we do not restate it.

6.4 Defining and Using Heap Patterns

Interesting programs organize heap data in structures such as linked lists, trees, graphs, and so on. To verify such programs, one needs to be able to specify and reason about heap structures. Since in matching logic the heap is just an algebraic term, it is natural to define heap patterns as terms of sort “heap” and then to axiomatize

them appropriately. Consider linked lists whose nodes contain two consecutive locations: an integer (data) followed by a pointer to next node (or 0, indicating the list end). One is typically interested in reasoning about the sequences of integers held by such list structures (and not about the particular pointers holding these data). It is then natural to define a list heap constructor “ $\text{list} : \text{Nat} \times \text{IntSeq} \rightarrow \text{Mem}$ ” taking a pointer (the location where the list starts) and a sequence of integers (the data held by the list, with ϵ for the empty sequence and $:$ for sequence concatenation) and yielding a fragment of memory; integer sequences can be easily axiomatized, for example as $\text{Seq}_\epsilon^+[\text{Int}]$. It does not make sense to define this as one would a function, since it is effectively non-deterministic, but it can be axiomatized as follows, in terms of patterns:

$$\begin{array}{c} \langle\langle\text{list}(p, \alpha) \otimes \sigma\rangle_{mem} \langle\varphi\rangle_{form} \langle X \rangle_{bnd} c\rangle \\ \Leftrightarrow \langle\langle\sigma\rangle_{mem} \langle p = 0 \wedge \alpha = \epsilon \wedge \varphi \rangle_{form} \langle X \rangle_{bnd} c\rangle \\ \vee \langle\langle p \mapsto [a, q] \otimes \text{list}(q, \beta) \otimes \sigma\rangle_{mem} \langle \alpha = a;\beta \wedge \varphi \rangle_{form} \langle X, a, q, \beta \rangle_{bnd} c\rangle. \end{array}$$

In words, a *list pattern* can be identified in the heap starting with pointer p and containing integer sequence α iff either the list is empty, so it takes no memory and its pointer is null (0), or the list is non-empty, so it holds its first element at location p and a pointer to a list containing the remaining elements at location $p + 1$. Using this axiom, one can prove properties about patterns, such as:

$$\begin{array}{c} \langle\langle 5 \mapsto 2 \otimes 6 \mapsto 0 \otimes 8 \mapsto 3 \otimes 9 \mapsto 5 \otimes \sigma \rangle_{mem} c\rangle \Rightarrow \langle\langle \text{list}(8, 3;2) \otimes \sigma \rangle_{mem} c\rangle \\ \langle\langle \text{list}(8, 3;2) \otimes \sigma \rangle_{mem} \langle X \rangle_{bnd} c\rangle \Rightarrow \\ \langle\langle 8 \mapsto 3 \otimes 9 \mapsto q \otimes q \mapsto 2 \otimes q+1 \mapsto 0 \otimes \sigma \rangle_{mem} \langle X, q \rangle_{bnd} c\rangle \end{array}$$

Figure 9 shows a snippet of a matching logic proof for list reverse, where we assume defined a conventional algebraic reverse operation $\text{rev} : \text{IntSeq} \rightarrow \text{IntSeq}$ on sequences of integers (easy to define equationally) and, as earlier in the paper, two patterns next to each other means that the former implies the latter. The loop invariant pattern (topmost one) specifies configurations in which: program variable \mathbf{p} is bound to pointer p , program variables \mathbf{x} and \mathbf{y} are bound to the same value x , and there are two disjoint lists in the heap, one starting with pointer p and holding sequence β and another starting with pointer x and holding sequence γ , such that $\text{rev}(\alpha) = \text{rev}(\gamma);\beta$, where α is the only free variable in the pattern (in addition to the “frames” ρ , σ , φ , and c , which are expected to always be free). There are two subproofs, one for the body of the loop (ending with the same invariant right before the “}”) and one for the desired property (list at \mathbf{p} will hold $\text{rev}(\alpha)$) after the loop. We cannot show all the details here, but there are several interesting uses of the list axiom, all discovered automatically by our prover in milliseconds. For example, since $x \neq 0$ in the first pattern in the loop body, one can expand $\text{list}(x, \gamma)$ in the heap and thus yield the second pattern in the loop body. The second pattern has all the configuration infrastructure needed to (symbolically) execute the four assignments. The resulting pattern can be applied the list axiom again to reshape its heap into one having a list at x ; the loop invariant follows then by an α -conversion of bound variables.

Below is an axiomatization of heap binary trees, assuming, like for lists, a corresponding mathematical notion of binary tree of integers (sort *IntTree* together with ϵ for the empty tree and with $a(l, r)$ for a tree with data a , left subtree l and right subtree r):

tree : $\text{Nat} \times \text{IntTree} \rightarrow \text{Mem}$

$$\begin{array}{c} \langle\langle\text{tree}(p, t) \otimes \sigma\rangle_{mem} \langle\varphi\rangle_{form} \langle X \rangle_{bnd} c\rangle \\ \Leftrightarrow \langle\langle\sigma\rangle_{mem} \langle p = 0 \wedge t = \epsilon \wedge \varphi \rangle_{form} \langle X \rangle_{bnd} c\rangle \\ \vee \langle\langle p \mapsto [a, l, r] \otimes \text{tree}(l, u) \otimes \text{tree}(r, v) \otimes \sigma \rangle_{mem} \\ \vee \langle t = a(u, v) \wedge \varphi \rangle_{form} \langle X, a, l, r, u, v \rangle_{bnd} c \rangle \end{array}$$

Using our matching logic prover in Section 7, we have verified several interesting properties over trees and several over trees combined with other heap structures, all available for download. For

```

< p ↦ p, x ↦ x, y ↦ x, ρ >env < list(p, β) ⊗ list(x, γ) ⊗ σ >mem
< rev(α) = rev(γ)β ∧ φ >form < p, x, β, γ >bnd c
while (x != 0) {
  < p ↦ p, x ↦ x, y ↦ x, ρ >env < list(p, β) ⊗ list(x, γ) ⊗ σ >mem
  < rev(α) = rev(γ)β ∧ x ≠ 0 ∧ φ >form < p, x, β, γ >bnd c
  < p ↦ p, x ↦ x, y ↦ x, ρ >env < list(p, β) ⊗ x ↦ [a, x'] ⊗ list(x', γ') ⊗ σ >mem
  < rev(α) = rev(γ)β ∧ γ = α:γ' ∧ φ >form < p, x, β, γ, x', γ' >bnd c
  y := [x+1]; [x+1] := p; p := x; x := y
  < p ↦ x, x ↦ x', y ↦ x', ρ >env < list(p, β) ⊗ x ↦ [a, p] ⊗ list(x', γ') ⊗ σ >mem
  < rev(α) = rev(γ)β ∧ γ = α:γ' ∧ φ >form < p, x, β, γ, x', γ' >bnd c
  < p ↦ x, x ↦ x', y ↦ x', ρ >env < list(x, αβ) ⊗ list(x', γ') ⊗ σ >mem
  < rev(α) = rev(γ')αβ ∧ φ >form < x, x', β, γ' >bnd c
  < p ↦ p, x ↦ x, y ↦ x, ρ >env < list(p, β) ⊗ list(x, γ) ⊗ σ >mem
  < rev(α) = rev(γ)β ∧ φ >form < p, x, β, γ >bnd c
}
< p ↦ p, x ↦ x, y ↦ x, ρ >env < list(p, β) ⊗ list(x, γ) ⊗ σ >mem
< rev(α) = rev(γ)β ∧ x = 0 ∧ φ >form < p, x, β, γ >bnd c
< p ↦ p, x ↦ 0, y ↦ 0, ρ >env < list(p, rev(α)) ⊗ σ >mem < φ >form < p >bnd c

```

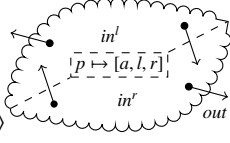
Figure 9. Matching logic proof snippet of list reverse.

example, we have verified a program traversing a tree in DFS and, as doing so, disposing each tree node and moving its element into a linked list; this program needed a stack-like heap structure in addition to lists and trees. Our most complex program that we verified automatically is, however, the (partial correctness of the) Schorr-Waite algorithm, which also needs a stack-like heap structure. The details are in Section 7 and on the tool website; here we only show our axiomatization of connected binary graphs in matching logic:

```

graph : Nat × Mem → Mem
graph : Nat × Mem × Mem → Mem
graph(p, g) = graph(p, g, ·)
<< graph(p, in, out) ⊗ σ > >mem < φ > >form < X > >bnd c
⇔ << σ > >mem < (p = 0 ∨ p ↦ a ⊆ out) ∧ in = · ∧ φ > >form < X, a > >bnd c
  < p ↦ [a, l, r] ⊗ graph(l, inl, out ⊗ inr ⊗ p ↦ [a, l, r]) > >mem
  < ⊗ graph(r, inr, out ⊗ inl ⊗ p ↦ [a, l, r]) > >mem
  < in = p ↦ [a, l, r] ⊗ inl ⊗ inr ∧ φ > >form < X, a, l, r, inl, inr > >bnd c

```



Like for lists and trees, $\text{graph}(p, g)$ is a heap pattern corresponding to a graph g rooted in p . Unlike for lists and trees, g is itself a heap structure. While for lists/trees it made sense to define abstract mathematical lists/trees (IntSeq and IntTree) aside and then to say that the heap list/tree encloses the mathematical object, for graphs we have no more abstract representation of a graph than its heap representation. Indeed, a graph in the heap is a set of location pairs $p \mapsto [l, r]$, where l and r are pointers to the left and the right neighbors of p . Hence, we define the graph heap construct to take a heap as its second argument.

To axiomatize $\text{graph}(p, g)$, we introduce and axiomatize a helping homonymous heap construct $\text{graph}(p, in, out)$ for *partial subgraphs*, that is subgraphs of the original graph that are not complete in that they may point out to nodes in the original graph, with the following intuition: in and out are subheaps of the original g ; in contains all the nodes $q \mapsto [l, r]$ that are in the corresponding subgraph; out , which is disjoint from in , contains nodes that the partial subgraph may point to; if $in \neq \cdot$ then p (the root) must be in in ; if $in = \cdot$ then p must be in out . Note that, indeed, $\text{graph}(p, g) = \text{graph}(p, g, \cdot)$, as defined above. Our axiom for partial graphs has three cases. The first two are when $p = 0$ or when p is in out , in which case the subgraph is empty. The third case is the interesting one and it says that there exists some split of the remaining nodes in in in two disjoint heaps, in^l and in^r , each corresponding to a partial subgraph potentially pointing out to nodes

in out , to p , or to nodes in the other partial subgraph. Note that the axiom covers all the cases, such as when $l = r$ or when one of l or r is p (one of the two subgraphs will be empty in these cases). There is a lot of non-determinism in how to split the partial subgraph in two partial subgraphs, but, as far as at least one such splitting exists, we are sure that all the nodes in the subgraph are reachable and that it only points out to nodes in out .

7. Proving Schorr-Waite With MATCHC

This section reports on the use of our matching logic program verifier MATCHC to verify the partial correctness of the Schorr-Waite algorithm. MATCHC is an evolving prototype which is being developed following the process described in this paper, but for the language C. Only a core subset of C is covered so far, called KERNELC [Roşu et al. 2009, Roşu and Schulte 2009], including all the features of HIMP but with `malloc` and `free` instead of `cons` and `dispose`, as well as C’s shortcut “boolean” constructs `&&`, `||`, etc. An additional *memory allocation table* cell is needed in the configuration to store the size of the allocated blocks, to know how many locations to deallocate with `free`. However, due to the modularity of matching logic and \mathbb{K} , that additional cell only affects the semantics of `malloc` and `free`. In particular, the definitions of heap pattern constructs and their axioms are not affected. Being C-specific, MATCHC takes a series of notational shortcuts to make it user-friendly and less verbose such as: the environment can always be inferred from context, so one can directly refer to the program variables in specifications (like in Hoare logic); one can define the parameter bound variables by simply prepending them with “?”; and one is not allowed to refer to the memory allocation table. Therefore, all one needs to define in patterns is the formula $\langle \varphi \rangle_{\text{form}}$ and heap $\langle \sigma \rangle_{\text{mem}}$ cells. We define them compactly, using C’s already existing `&&` construct, writing the σ in brackets. For example, the list reverse loop invariant in Figure 9 (first pattern there) is written as follows in MATCHC (in plain text, we write `**` instead of \otimes):

```
inv [list(p, ?B) ** list(x, ?C) ** rest] && rev(A) == rev(?C) : ?B
```

We prepend invariant assertions with keyword `inv`; we additionally allow annotation keywords `assert` and `assume`, the latter being useful both to assume the original pre-condition and to eliminate functions. One can actually use any C expressions within MATCHC annotations, not only program variables; these are evaluated in the current configuration to compute the actual asserted/assumed pattern. For example, one can write assertions like

```
assert [list(p, ?alpha) ** rest] && (*p > 0 || *(p+1) != 0)
```

MATCHC currently only supports heap patterns specified like in Section 6.4, that is, of the form “heap pattern iff cases”. These axioms are applied lazily. They are applied from left to right only when a heap access is attempted and the desired location is not available, as in the case of evaluating $*(p + 1)$ while the current heap includes the term $\text{list}(p, ?\alpha)$. We call this process *heap derivation* and is similar to rearrangement rules of Berdine et al. [2005, Sec. 3.2] and the “focus” step of shape analysis [Sagiv et al. 2002]. The axioms are applied from right to left only when one asserts a pattern that the current heap does not match. Since right-to-left applications of axioms are well-founded (they reduce the size of the configuration), we apply them exhaustively. MATCHC gets stuck when no derivation can provide the desired location or when the asserted pattern cannot be matched. One particularly useful feature of MATCHC in practice is that it gets stuck on exactly the statement or expression that cannot be handled. MATCHC is available for download [Ellison and Roşu 2009], together with examples including programs using lists, trees, queues, stacks, and graphs. Like Caduceus, MATCHC can be connected to various SMT

```

//@ assume [cleanGraph(root, in)]
t=root; p=null ;
//@ inv t==null && [stackInGraph(p, in)]
  || t!=null &&
    ( *t==1 && [markedGraph(t, ?in_t, ?in_p) **
      stackInGraph(p, ?in_p, ?in_t)]
      || *t==0 && [ cleanGraph(t, ?in_t, ?in_p) **
        stackInGraph(p, ?in_p, ?in_t)]
    ) && in == ?in_t ** ?in_p
while (p!=null || t!=null && *t==0) {
  if (t==null || *t==1) {
    if (*t==1) {q=t; t=p; p=*(p+3); *(t+3)=q;}
    else {q=t; t=*(p+3); *(p+3)=*(p+2); *(p+2)=q; *(p+1)=1;}
  } else {q=p; p=t; t=*(t+2); *(p+2)=q; *p=1; *(p+1)=0;}
}
//@ assert [markedGraph(t, in)]

```

Figure 10. The Schorr-Waite graph marking algorithm.

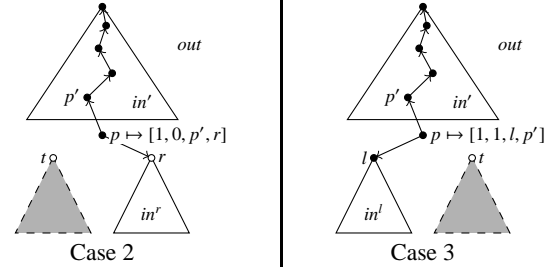
solvers. Unlike other provers, since MATCHC is implemented using rewriting, it can take advantage of rewrite rule simplifications that are applied on the fly, wherever they match. In many cases the SMT solvers need not even be called.

In the remainder of this section we discuss how MATCHC proves Schorr-Waite. We assume the reader is familiar with the algorithm. We borrowed the C code from Hubert and Marché [2005], replacing record accesses with explicit memory accesses and bookkeeping bits with locations (MATCHC does not support structures and bit accesses yet); if t points to a node in the graph, then $*t$ is the bit saying whether that node is marked or not (0 clean, 1 marked), $*(t+1)$ is the bit saying whether the left (0) or the right (1) neighbor is next to be explored, and $*(t+2)$ and $*(t+3)$ point to the left and the right neighbor, respectively. Figure 10 shows the annotated program. MATCHC verifies it in about 16 seconds on a Linux 2.5GHz machine, analyzing a total of 227 cases. The pre-condition states that there is a clean graph in starting at pointer $root$ in the heap and the post-condition states that at the end of the execution the heap contains that same graph in , but marked.

We define `cleanGraph` and `markedGraph` almost exactly as the graph heap pattern in Section 6.4, the only difference being that the “ $p \mapsto [l, r]$ ” term in the heap of the second case is replaced by “ $p \mapsto [0, 0, l, r]$ ” for `cleanGraph` and by “ $p \mapsto [1, 1, l, r]$ ” for `markedGraph`. This way, a term `cleanGraph(t, in, out)` in the heap (with in and out disjoint) corresponds to a heap representation of the partial graph in in which all nodes have the bookkeeping bits 0 (i.e., they are “clean”), are reachable from t , and any other node reachable from t which is not in in must be in out .

The algorithm maintains two pointers: t points to a partial subgraph to be processed next, and p points to a stack-in-the-graph which can reach all the nodes in the original graph that are not in the partial subgraph at t . The partial subgraph at t is either clean, in which case it needs to be visited, or is marked, in which case we swing or backtrack. Since there could be multiple ways to reach the same node, there is a high degree of non-determinism in choosing the partial subgraph at t ; all that matters is that the remaining nodes are reachable from p . In other words, if $?in_t$ is the partial subgraph at t and $?in_p$ is the partial subgraph corresponding to the stack at p , then the two partial subgraphs should form a partition of the original graph, that is, “ $in == ?in_t ** ?in_p$ ”.

Before we define our critical heap pattern `stackInGraph`, let us first intuitively discuss the invariant. There are two top level cases. If $t == \text{null}$ (when a leaf is reached) then the stack should contain the entire graph. If $t != \text{null}$ then there are two cases again: if t points to a clean graph, then the heap should be partitionable into a clean graph at t and a stack at p , and if t points to a marked graph



`stackInGraph` : $Nat \times Mem \rightarrow Mem$
`stackInGraph` : $Nat \times Mem \times Mem \rightarrow Mem$

$$\begin{aligned}
& \text{stackInGraph}(p, g) = \text{stackInGraph}(p, g, \cdot) \\
& \langle \langle \text{stackInGraph}(p, in, out) \otimes \sigma \rangle_{mem} \langle \varphi \rangle_{form} \langle X \rangle_{bnd} c \rangle \\
& \Leftrightarrow \langle \langle \sigma \rangle_{mem} \langle p = 0 \wedge in = \cdot \wedge \varphi \rangle_{form} \langle X \rangle_{bnd} c \rangle \\
& \vee \left\langle \left\langle \begin{aligned} & p \mapsto [1, 0, p', r] \otimes \text{stackInGraph}(p', in', out \otimes in' \otimes p \mapsto [l, r]) \\ & \otimes \text{cleanGraph}(r, in', out \otimes in' \otimes p \mapsto [l, r]) \end{aligned} \right\rangle_{mem} \right\rangle \\
& \left\langle in = p \mapsto [l, r] \otimes in' \otimes in' \wedge \varphi \right\rangle_{form} \langle X, p', l, r, in', in' \rangle_{bnd} c \\
& \vee \left\langle \left\langle \begin{aligned} & p \mapsto [1, 1, l, p'] \otimes \text{stackInGraph}(p', in', out \otimes in' \otimes p \mapsto [l, r]) \\ & \otimes \text{markedGraph}(l, in', out \otimes in' \otimes p \mapsto [l, r]) \end{aligned} \right\rangle_{mem} \right\rangle \\
& \left\langle in = p \mapsto [l, r] \otimes in' \otimes in' \wedge \varphi \right\rangle_{form} \langle X, p', l, r, in', in' \rangle_{bnd} c
\end{aligned}$$

Figure 11. Axiom for heap pattern `stackInGraph`.

then the heap should be partitionable into a marked graph at t and a stack at p . We believe that this invariant is as simple as it can be.

Figure 11 shows our axiomatization for `stackInGraph`. There are three cases to distinguish, but before discussing those let us clarify the meaning of `stackInGraph(p, in, out)`. Arguments in and out will always be partial subgraphs of the original graph, so they are unaffected by algorithmic mutilations of the graph. Then `stackInGraph(p, in, out)` corresponds to a Schorr-Waite style stack structure in the heap that starts with p , reaches all the nodes in the original partial subgraph in , and can potentially point out to nodes in out . The first case states that the stack is empty iff p is null and the corresponding partial subgraph is empty. The second and third cases are similar, so we only discuss the third: if p points to a marked node (the first “1” in “[1, 1, l, p’]”) whose left neighbor at l was already visited (the second “1” in “[1, 1, l, p’]”) and whose current right neighbor is p' , then it must be the case that we can partition the current stack into a marked partial subgraph at l , the node p in the original graph, and a (smaller) stack at p' . The pictures in Figure 11 show these two cases (the grey subgraph at t is not in in ; it is there only to better relate the stack to the algorithm).

As mentioned, there are 227 cases to analyze. Let us only informally discuss one of them. Recall from Figure 6 that, when verifying the while loop, our \mathbb{K} prover checks the invariant when it first reaches it, then assumes it and generates two cases; one of them executes the loop body and then asserts the invariant. Our invariant generates itself several cases. Let us consider the one in which $t != \text{null}$ and $*t = 1$, that is, the heap has the form `[markedGraph(t, ?in_t, ?in_p) ** stackInGraph(p, ?in_p, ?in_t)]` for some symbolic $?in_t$ and $?in_p$. In the loop body, consider the “swing” case when $*(p+1) == 0$. To pass the conditional’s $*(p+1) == 1$ guard, the stack axiom needs to be applied from left-to-right to derive the stack term. Only the second case is feasible for the “swing” branch, which expands the stack into $p \mapsto [1, 0, p', r] \otimes \text{stackInGraph}(p', in', ?in_t \otimes in' \otimes p \mapsto [l, r]) \otimes \text{cleanGraph}(r, in', ?in_t \otimes in' \otimes p \mapsto [l, r])$ for some symbolic p', l, r, in', in' . Now the “swing” branch can also be executed, because it only affects the portion of heap $p \mapsto [1, 0, p', r]$, transforming it into $p \mapsto [1, 1, l, p']$; the environment will also hold the map-

pings $q \mapsto t$ and $t \mapsto r$, where t is original symbolic value of t in the assumed environment. One can now apply the stack axiom from right-to-left grouping the original marked graph at t , the stack at p' and the locations $p \mapsto [1,1,t,p']$, into the term $\text{stackInGraph}(p, in \otimes ?in.t \otimes p \mapsto [l,r, in'])$. The symbolic heap now has a stack at p and a clean graph at r , which satisfies the asserted invariant. Indeed, since $t \mapsto r$ in the environment, the only feasible path in the invariant is the one containing a stack and a clean graph; the partitionability requirements of the involved partial subgraphs can be easily checked. This case, as well as the other similar 226 cases, can all be dispatched and verified automatically by MATCHC in 16 seconds on a conventional Linux machine.

8. Related Work

Matching logic is most closely related to Hoare logics, separation logic, and shape analysis.

There are many Hoare logic verification frameworks, such as the KeY project [Beckert et al. 2007] and ESC/Java [Flanagan et al. 2002] for Java, as well as the Spec# tool [Barnett et al. 2004] for C#, and HAVOC [Lahiri and Qadeer 2006, Hackett et al. 2008], and VCC [Cohen et al. 2009] for standard C. Caduceus, part of the Why platform [Filliâtre and Marché 2004, 2007, Hubert and Marché 2005], has seen much success with the first order logic approach, including proving many correctness properties relating to the Schorr-Waite algorithm. However, their proofs were not entirely automated. The weakness of traditional Hoare-like approaches is that reasoning about non-inductively defined data-types and about heap structures tend to be difficult, requiring extensive manual intervention in the proof process.

Separation logic [O'Hearn and Pym 1999, Reynolds 2002] is an extension of Hoare logic. There are many variants and extensions of separation logic which we do not discuss here, but we'd like to mention that there is a major difference between separation and matching logic: the former attempts to extend and "fix" Hoare logic to work better with heaps, while matching logic attempts to provide an alternative to Hoare logics in which the program configuration structure is explicit in the specifications, so heaps are treated uniformly just like any other structures in the configuration. Smallfoot [Berdine et al. 2005], Verifast [Jacobs and Piessens 2008], and jStar [Distefano and Parkinson 2008] are separation logic tools; as far as we know, they have not proven Schorr-Waite — these tools have good support for proving memory safety, though.

Shape analysis [Sagiv et al. 2002] allows one to examine and verify properties of heap structures. It has been shown to be quite powerful when reasoning about heaps, leading to an automated proof of total correctness for the Schorr-Waite algorithm [Loginov et al. 2006] on binary trees. The ideas of shape analysis have also been combined with those of separation logic [Distefano et al. 2006] to quickly infer invariants for programs operating on lists.

Other notable frameworks for reasoning about heap structures include Møller and Schwartzbach [2001], Bozga et al. [2003], McPeak and Necula [2005], Rinetzky et al. [2005] and Mehta and Nipkow [2005], which we do not have space to discuss in detail.

Dynamic logic [Harel et al. 1984] also works with pairs instead of triples, embedding the code in the specification.

Meseguer and Roşu [2007] give a brief introduction to \mathbb{K} and the manuscript [Roşu 2007] gives a detailed presentation of it and comparisons with other language definitional formalisms. In short, \mathbb{K} is related to: reduction semantics (with [Wright and Felleisen 1994] and without [Plotkin 2004] evaluation contexts) but it is context insensitive and unconditional so it can be executed on existing rewrite engines; the SECD [Landin 1964] and other abstract machines, but it is also denotational in that a \mathbb{K} definition is a mathematical theory with initial model semantics; the CHAM [Berry and Boudol 1992], but it allows a higher degree of concurrency and

it is efficiently executable; continuations [Reynolds 1993], but the novice needs not be aware of them; refocusing [Danvy and Nielsen 2004], but it is not aimed at "implementing" evaluation contexts, can deal with non-deterministic context grammars, and its refocusing steps are reversible; etc. \mathbb{K} has been used in programming language courses for more than five years and in research projects to define a series of existing programming languages (like Java 1.4, Scheme, C) and for prototyping many paradigmatic languages. The URL <http://fs1.cs.uiuc.edu/k> gives more detail on \mathbb{K} , including several papers and a prototype tool that takes \mathbb{K} definitions like the one in Figure 2 and generates Maude [Clavel et al. 2007] modules that can be executed and formally analyzed (e.g., model-checked) using the available Maude tools.

9. Conclusion and Future Work

We showed that an executable rewriting logic semantics (RLS) of a language can be turned into a provably correct and executable matching logic verifier with relatively little effort. As shown in our companion report [Ellison and Roşu 2009], one can conservatively associate a matching logic proof system to any Hoare logic proof system, so matching logic is at least as expressive as Hoare logic. Moreover, since matching logic specifications have access to the structure of the program configuration, it is relatively easy to systematically axiomatize complex heap structures. Consequently, our results not only bridge the gap between formal language semantics and program verification, but are also practical. As a case study showing the feasibility of our approach, we have automatically verified the partial correctness of the challenging Schorr-Waite algorithm using our MATCHC prover built on top of Maude.

Matching logic is very new (it was introduced this year in a technical report [Roşu and Schulte 2009]), so there is much work left to be done. For example, we would like to extend MATCHC to verify multithreaded programs. The intrinsic separation available in matching logic might simplify verifying shared resource access. Another interesting investigation is to infer pattern loop invariants; since configurations in our approach are just ground terms that are being rewritten by semantic rules, and since patterns are terms over the same signature with constrained variables, we believe that narrowing and/or anti-unification can be good candidates to approach the problem of invariant inference.

Since our matching logic verification approach makes language semantics practical, we believe that it will stimulate interest in giving formal rewrite logic semantics to various programming languages. We have started implementing a \mathbb{K} front-end to Maude that allows one to define semantics of languages quickly and compactly.

Idea: To do: prove that *ML-false* and *ML-case* are derived rules, that is, that one can prove with the other rules alone what one can prove with these.

Idea: To do: prove that if $\Gamma \Rightarrow \Gamma_1 \vee \Gamma_2$ and all three are satisfiable, then they must have the same computation; moreover, one can replace that computation by anything else and the implication above still holds. also, show that Γ is unsatisfiable iff it is of the form $\langle\langle \text{false} \rangle_{\text{form}} c \rangle$.

References

- M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system. In *CASSIS'04*, volume 3362 of *LNCS*, pages 49–69, 2004.
- M. Barnett, B. yuh Evan Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO'05*, volume 4111 of *LNCS*, pages 364–387, 2006.
- B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. 2007.
- J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic exec. with separation logic. In *APLAS'05*, volume 3780 of *LNCS*, pages 52–68, 2005.

- G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.
- M. Bozga, R. Iosif, and Y. Laknech. Storeless semantics and alias logic. In *PEPM’03*, pages 55–65. ACM, 2003.
- M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *LNCS*. 2007.
- E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Research, 2009.
- O. Danvy and L. Nielsen. Refocusing in reduction semantics. Technical Report RS-04-26, BRICS, 2004.
- D. Distefano and M. J. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA’08*, pages 213–226. ACM, 2008.
- D. Distefano, P. W. O’Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS’06*, pages 287–302, 2006.
- C. Ellison and G. Roşu. Matching logic website – Tools and examples, 2009. URL <http://fs1.cs.uiuc.edu/ml>.
- A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal analysis of Java programs in JavaFAN. In *CAV’04*, volume 3114 of *LNCS*, pages 501–505, 2004.
- J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. *Formal Methods and Software Engineering*, pages 15–29, 2004.
- J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV’07*, volume 4590 of *LNCS*, pages 173–177, 2007.
- C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI’02*, pages 234–245. ACM, 2002.
- J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic specification in action*, pages 3–167. Kluwer, 2000.
- B. Hackett, S. Lahiri, S. Qadeer, and T. Ball. Scalable modular checking of system-specific properties: Myth or reality? Technical Report MSR-TR-2008-82, Microsoft Research, 2008.
- D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604. MIT, 1984.
- T. Hubert and C. Marché. A case study of C source code verification: The Schorr-Waite algorithm. In *SEFM’05*, pages 190–199. IEEE, 2005.
- B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Katholieke Universiteit Leuven, 2008.
- S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL’06*, pages 115–126. ACM, 2006.
- P. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
- A. Loginov, T. Reps, and M. Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *SAS’06*, 2006.
- S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *CAV’05*, volume 3576 of *LNCS*, pages 476–490, 2005.
- F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information & Computation*, 199(1-2):200–227, 2005.
- J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
- J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.
- A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. *SIGPLAN Not.*, 36(5):221–231, 2001.
- P. W. O’Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5:215–244, 1999.
- G. Plotkin. A structural approach to operational semantics. *Journal of Logic & Algebraic Programming*, 60-61:17–139, 2004.
- J. C. Reynolds. The discoveries of continuations. *Lisp & Symbolic Computation*, 6(3-4):233–248, 1993.
- J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS’02*, pages 55–74. IEEE, 2002.
- N. Rinetzky, J. Bauer, T. W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, pages 296–309. ACM, 2005.
- G. Roşu. K: A rewriting-based framework for computations – Preliminary version. Technical Report UIUCDCS-R-2007-2926, University of Illinois, Department of Computer Science, 2007.
- G. Roşu and W. Schulte. Matching logic – Extended report. Technical Report UIUCDCS-R-2009-3026, Univ. of Illinois, 2009.
- G. Roşu, W. Schulte, and T. Şerbănuţă. Runtime verification of C memory safety. In *RV’09*, LNCS, 2009. to appear.
- M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
- T. F. Şerbănuţă, G. Roşu, and J. Meseguer. A rewriting logic approach to operational semantics. *Information & Computation*, 207:305–340, 2009.
- A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information & Computation*, 115(1):38–94, 1994.